Trustworthy Variant Derivation with Translation Validation for Safety Critical Product Lines[☆]

Alexandru F. Iosif-Lazăr[1], Andrzej Wąsowski[1]

IT University of Copenhagen

Abstract

Software product line (SPL) engineering facilitates development of entire families of software products with systematic reuse. Model driven SPLs use models in the design and development process. In the safety critical domain, validation of models and testing of code increases the quality of the products altogether. However, to maintain this trustworthiness it is necessary to know that the SPL tools, which manipulate models and code to derive concrete product variants, do not introduce errors in the process.

We propose a general technique of checking correctness of product derivation tools through translation validation. We demonstrate it using Featherweight VML–a core language for separate variability modeling relying on a single kind of variation point to define transformations of artifacts seen as object models. We use Featherweight VML with its semantics as a correctness specification for validating outputs of a variant derivation tool. We embed this specification in the theorem proving system Coq and develop an automatic generator of correctness proofs for translation results within Coq. We show that the correctness checking procedure is decidable, which allows the trustworthy proof checker of Coq to automatically verify runs of a variant derivation tool for correctness.

We demonstrate how such a simple validation system can be constructed, by using this to validate variant derivation of a simple variability model implementation based on the Eclipse Modeling Framework. We hope that this presentation will encourage other researchers to use translation validation to validate more complex correctness properties in handling variability, as well as demonstrate to commercial tool vendors that formal verification can be introduced into their tools in a very lightweight manner.

[☆]This article is a full version of the extended abstract presented at the 25th Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn.

Email addresses: afla@itu.dk (Alexandru F. Iosif-Lazăr), wasowski@itu.dk (Andrzej Wąsowski)

[1]Supported by ARTEMIS JU under grant agreement n° 295397 and by Danish Agency for Science, Technology and Innovation

1. Introduction

Variability modeling in software product lines. Model-driven development [1] of software products employ models to represent the product architecture. When several products share a common set of core assets they can be developed as a software product line [2]. Modeling the product line architecture as a single *base model* facilitates the *derivation* of new *product variants* by reusing artifacts from existing ones. Variability models describe how the artifacts can be selected and recombined into new products.

Separate variability models are independent of the language in which the base model is developed so they can be reused to some extent to handle a system's variability at multiple development phases. The range of distinguishing characteristics which vary among the products of a product line is specified with feature models [3] (or alternatives such as decision models [4]) Each individual product is described by selecting a set of features thus creating a particular configuration. Constraints and dependencies between features are often specified to determine which configurations are valid. Features are realized by implementation artifacts (e.g. formal specifications, object models, source code) contained in a base model. Thus, we need both a mapping from the feature model to the base model and a process called *variant derivation* through which the artifacts can be selected and recombined into new product variants.

The *Orthogonal Variability Model* (OVM) [5], *Delta Modeling* [6] and the *Common Variability Language* (CVL) [7] are examples of separate variability modeling languages.

Trustworthy variant derivation tools. There is a great variety of tools that implement variability modeling languages and facilitate variant derivation. Trustworthy variant derivation is essential to the development of safety critical embedded systems in domains such as automotive or industrial automation [8, 9]. Industrial standards such as IEC 61508 [10] mandate the use of state of the art tools and quality assurance techniques. So far, the industry certifies individual products, or even avoids introducing any variability into safety critical parts of the systems[1].

Trustworthy variant derivation has two requirements. The first is verifying the product line base model, the variability model and the configuration model trying to identify and report errors introduced by the model designers. The second requirement is verifying the product variant derivation tool to ensure that the derivation process is implemented and executed correctly. While there is a need for good verification methods for both input models and tools, most of the research is focused on the former. The tools that implement variability modeling and variant derivation are usually assumed to be correct. This is partly because the tools employ complex algorithms and depend on external libraries which makes it impossible to formally verify them. Nevertheless, qualification is required for code manipulation tools (such as variant derivation) used in

[1] Personal communication with partners in ARTEMIS projects.

producing code influencing functional safety functions. Our goal is to provide a non-intrusive way of verifying that the output of these tools is produced as prescribed by the input models and to enable usable qualification strategies. We achieve this through *translation validation* [11].

Translation validation recognizes that it might be too challenging to verify a translator (originally a compiler; in our case a variant derivation tool). After all, verifying a translator to be correct once for all, means verifying that it will behave correctly on all possible inputs, which is usually an infinite set with complex properties. In practice, a translator will never be run on the entire set, but on a finite subset. Consequently, it seems wasteful to verify its correctness once for all inputs. With translation validation we do not validate the translator itself, but the output of each execution.

The approach is entirely automatic. Usually the translator is extended to generate a formal proof of correctness of the output with respect to the input. This proof is then checked automatically using an independent proof checker. In the usual scenario, where no bug of translation is uncovered, both tools succeed automatically. In the unlikely case of the translator failing to generate the proof (due to a possible bug) or the proof checker reporting that the proof is incorrect, the user can be warned about the error. This is less convenient than eliminating errors altogether, but prevents the use of erroneous output in a critical system, so the harm is avoided.

The main benefit of this validation method is that developing a translation validator is much simpler than verifying the entire translator. Even a simple variant derivation tool, as discussed in this paper, relies on a number of complex frameworks and libraries (Eclipse Modeling Framework, XML libraries, standard libraries of the programming language, the programming language itself, etc). Building a formal model of all these elements is unbelievably laborious, whereas translation validation only requires providing a semantic based argument that the output structure is correct with respect to the input structure, independently of how complex the frameworks used in the process are. Thus we believe that translation validation is a viable way of increasing trustworthiness of commercial software tools. In the paper we demonstrate how translation validation can be implemented for variant derivation as an add-on, with minimal changes to the implementation of the tool performing variant derivation.

The translation validation approach is independent of the actual implementation. What it does require is: (i) a common semantic framework for both the input and the output; (ii) a formalization of the notion of *correct execution* (iii) a proof method which, based on the input, allows to automatically verify that the output is correct.

Contributions. In this paper we realize the translation validation approach for an abstract variability modeling language, that is able to capture abstractions of executions of many of the above mentioned modeling notations, in particular of those that subscribe to a separate variability perspective (although the translation validation method in itself does not require separate specification of variability, this is how we scope our demonstration). Our contributions are:

- A core language for separate variability modeling, Featherweight VML, along with an abstract semantics, which is as expressive and versatile as other existing variability modeling languages. We will use it to represent abstractions of concrete variability models.

- A formal specification of semantics of Featherweight VML, a prerequisite for building a translation validation tool. This captures semantics of relations between features with cardinalities [12] and the base model by copying and flattening the variability model. We also provide a copying semantics for the variant derivation process. We define *two* simple rules for determining which model elements are part of the desired product variant. Compared to in-place model transformations, a copying semantics can more easily be implemented in declarative rule-based model transformation languages and it is easier to reason about using theorem provers. To the best of our knowledge, variability models including both cardinality-based feature modeling and a mapping to implementation artifacts have not been formalized so far.

- A confluence result for our semantics: while other approaches to defining semantics of separate variability modeling languages suggest an implementation by in-place transformations (which makes the order of transformations critical) our rules always produce the same result, independently of the order in which they are applied. Incidentally, this opens for new opportunities to implement variant derivation tools using graph transformations.

- An embedding of the above definitions into the semantic framework of the Coq theorem prover, including a formal mechanically checkable proof of correctness of the embedding.

- A proof of concept translation validation tool for Featherweight VML using the above embedding. Our translation validation strategy is black-box, so it does not require modification of an existing variant derivation tool (in our case a custom in-house tool based on the Eclipse Modeling Framework).

Due to use of the abstraction, our method does not require meticulous formalizations of all the aspects of the variability modeling. The approach allows incremental development. In our demonstration, we instantiate the idea only for connectivity properties of the base model, which keeps the development cost low. If more properties need to be tracked they can be added in subsequent iterations by enriching Featherweight VML and the abstractions.

The paper proceeds as follows. Section 2 provides an analysis of different variability modeling languages in order to determine the core requirements. It also introduces an example of a software product line. Section 3 presents our approach to translation validation. Section 4.1 introduces a minimal representation of base models as graphs. Sections 4.2 and 4.3 describe the formal syntax and semantics of Featherweight VML. Section 5 contains the implementation of Featherweight VML in Coq and describes the validation of a black-box demo tool. We discuss

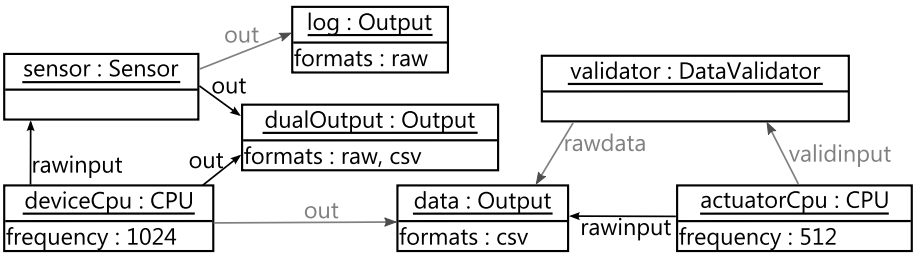

Figure 1: Example base model for a product line of devices

the advantages and limitations of translation validation and also the related work in Sec. 6 and we conclude in Sec. 7.

2. Core Requirements for Variability Modeling

In order to develop a generic method for validating the correctness of variant derivation tools, we require a versatile foundation for variability modeling. To this end, we compare CVL, Delta Modeling and OVM, aiming to find similarities in the way these languages represent and execute variability models over system models. The results help us setting a foundation for defining the syntax and semantics of Featherweight VML. Later we will use Featherweight VML to represent abstractions of variability models during their executions.

2.1. A running example

In Fig. 1 we introduce an example of a product line architecture from which several product variants can be derived. This example will help illustrate the characteristics of OVM, Delta Modeling, CVL and Featherweight VML.

Our system represents a safety critical monitoring device. A minimal product variant is composed of the deviceCpu, the sensor and one of the possible outputs. The deviceCpu receives raw input from the sensor and relays the data as comma-separated values (csv) to the dualOutput to which the sensor also sends the raw data.

Alternatively, sensor may output raw data directly to a log as represented by the gray link out. Yet another possibility is for the deviceCpu to send csv data to a data output which serves as input for an actuatorCpu. As an extra check, the actuatorCpu may compare the raw data with validated input from a validator.

Usually modeling tools require that all elements are contained in an object representing the model root. To keep figures and explanations simple, we ignore the root object in our examples.

2.2. Overview of Variability Modeling Languages

The Orthogonal Variability Model. OVM [5] is designed to handle variability between products. It leaves aside the common parts. It uses *variation points* to specify which characteristics can vary (e.g. color) and *variants* to specify how

they vary (e.g. red, blue etc.). Dependency relations between the variation points and variants limit the set of valid configurations. All artifacts are contained in a single model. Both variation points and variants are mapped directly to these artifacts so the solution space does not involve complex transformations. When a configuration is selected (the desired variants are selected for each variation point) the variability model is executed by extracting only the artifacts that the configuration refers to.

In our example, the deviceCpu and the sensor are part of all possible products, thus they do not make the object of the OVM. Instead they are included in all products by default. A variation point VP_1 is needed to specify that the output of the system can vary. This variation point has three variants: V_1 is realized by the dualOutput, V_2 is realized by the log and V_3 is realized by the data and actuatorCpu objects together. Since having both a dualOutput and a log or data output is redundant, we can make the variant pair (V_1, V_2) as well as (V_1, V_3) mutually exclusive.

Another variation point VP_2 specifies that the validator object can vary by having a single optional variant. VP_2 would be dependent of the selection of the variant V_3 for VP_1 since the existence of the validator only makes sense if the data and actuatorCPU are also included in the product. For brevity, we have omitted to show how the variants are referring to the links in the base model (i.e. out, rawinput, rawdata, validinput). Nonetheless, the OVM variants should refer to all artifacts that must be included in the final products.

Delta Modeling. In *Delta Modeling* [6], a product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing artifacts. Delta Modeling can use any flavour of feature model. Each delta module has an application condition which the configuration must respect in order for the delta to be executed. Delta Modeling can be applied to textual languages, such as the HATS Abstract Behavioral Specification Language [13], or graphical modeling languages, such as Matlab/Simulink [14]. Recently [15], a method has been proposed to systematically deriving a delta language from the grammar of a given base language. Even though Delta Modeling syntax is highly adaptable to the language of the base model, its main concepts remain the same regardless of the implementation.

Listing 1 shows how we can use Delta Modeling to handle the variability in our example. It begins by specifying a list of **features** and a constraint for valid **configurations**. The **core** and **delta** modules add and remove objects and links. The **core** module adds the objects deviceCpu, sensor and dualOutput identified by the object name. It also adds the rawinput link from deviceCpu to sensor and the two out links from deviceCpu and sensor to dualOutput. It is executed when the Dual feature is selected.

The rest of the **delta** modules add and remove objects and links to express the alternative product variants. The **delta** DValidator is executed if both Actuator

and Validator features have been selected, but only after the execution of the **delta** DActuator, as specified with the **when** and **after** clauses.

```
1  features Dual, Log, Actuator, Validator
2  configurations Dual ⊕(Log ∨(Actuator ∨(Actuator ∧ Validator)))
3
4  core Dual {
5    adds objects {deviceCpu, sensor, dualOutput}
6    adds links {deviceCpu.rawinput -> sensor, deviceCpu.out -> dualOutput,
7      sensor.out -> dualOutput}
8  }
9
10 delta DLog when Log {
11   removes objects {dualOutput}
12   removes links {deviceCpu.out -> dualOutput, sensor.out -> dualOutput}
13   adds objects {log}
14   adds links {sensor.out -> log}
15 }
16
17 delta DActuator when Actuator {
18   removes objects {dualOutput}
19   removes links {deviceCpu.out -> dualOutput, sensor.out -> dualOutput}
20   adds objects {data, actuatorCpu}
21   adds links {deviceCpu.out -> data, actuatorCpu.rawinput -> data}
22 }
23
24 delta DValidator when (Actuator ∧ Validator) after DActuator {
25   adds objects {validator}
26   adds links {validator.rawdata -> data, actuatorCpu.validinput -> validator}
27 }
```

Listing 1: Delta model of the device product line

The Common Variability Language. CVL [7] is an industrial attempt to create a generic language that facilitates separate variability modeling for base models specified in any MOF-based language [16]. It employs specialized features called *variability specifications* which can be resolved in particular ways: choices require a yes/no resolution; variables require a value for a specific artifact; classifiers represent features that can be instantiated multiple times in a configuration (similar to features with cardinalities [12]). CVL uses a constraint language to specify constraints over the variability specification tree. Configurations are represented as *resolution models*.

The variability specifications are realized by artifacts which can be manipulated through a wide range of transformations called *variation points*[2] Among the most common variation points are object/link existence, object substitution

[2]CVL and OVM variation points are different concepts.

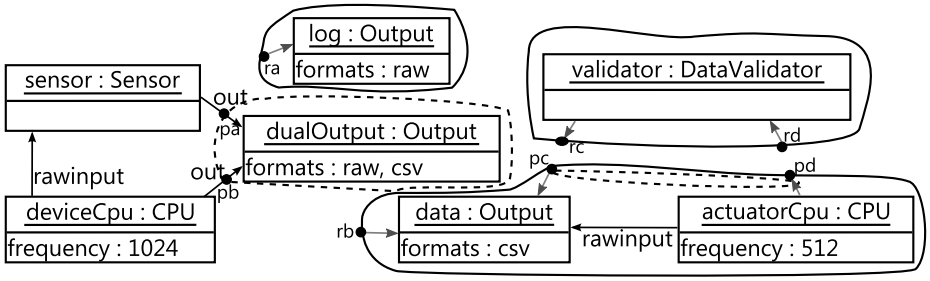

Figure 2: CVL model of the device product line

(with another object), link-end substitution (substitutes one endpoint with another) or value assignment for object variables. However, most variation points are syntactic sugar as they can be expressed using the fragment substitution. This variation point can replace an entire fragment of the model with another fragment (possibly defined in a separate library).

We illustrate this using our running example. We begin with a base model shown in Fig. 2 where it is considered that the objects deviceCpu, sensor and dualOutput and the links between them are included by default. Fragment substitutions are used to specify additional transformations. A fragment substitution is similar to a delta module. It consists of a *placement* fragment (surrounded by dashed lines) containing the elements that will be removed and a *replacement* fragment (surrounded by solid lines) containing the elements that will be inserted instead. CVL defines fragments by surrounding them with an imaginary closed curve and placing boundary points whenever links cross the curve. Boundary points are elements that fully define all references going in and out of placement/replacement fragments.

To show that dualOutput must be removed we require the boundary points pa and pb pointing to the placement fragment. The boundary point ra marks the entry to the replacement fragment composed of the log object. A binding between pa and ra indicates that the link targeting dualOutput should target log. The actual elements of a fragment are discovered by traversing the model from the entry boundary points. The fragments are fully discovered when all connected elements have been reached (the direction of the links is ignored) or when the traversal has been cut off by other boundary points. Traversing the model from rb will retrieve the replacement fragment composed of data, actuatorCpu and the link between them. The following fragment substitutions specify the possible changes to the base model:

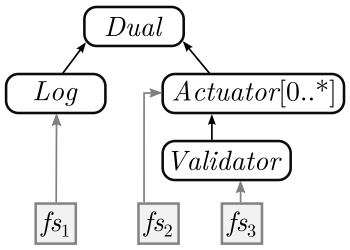

Figure 3: A variability specification tree

$$\mathsf{fs}_1\{\ \mathit{placement}\{\mathsf{pa},\mathsf{pb}\}\ \mathit{replacement}\{\mathsf{ra}\}\ \mathit{binding}\{(\mathsf{pa},\mathsf{ra})\}\ \}$$

$\mathsf{fs}_2\{$ *placement*$\{\mathsf{pa}, \mathsf{pb}\}$ *replacement*$\{\mathsf{rb}\}$ *binding*$\{(\mathsf{pb}, \mathsf{rb})\}$ $\}$

$\mathsf{fs}_3\{$ *placement*$\{\mathsf{pc}, \mathsf{pd}\}$ *replacement*$\{\mathsf{rc}, \mathsf{rd}\}$ *binding*$\{(\mathsf{pc}, \mathsf{rc}),\ (\mathsf{pd}, \mathsf{rd})\}$ $\}$

Finally, CVL organizes the features in a variability specification tree and binds the fragment substitutions accordingly as shown in Fig. 3. In this example the *Actuator* is a VClassifier meaning that fs_2 can be executed multiple times to obtain multiple actuators.

2.3. Comparative Analysis

Modeling the variations and the configurations is done for all three variability modeling languages using some form of feature models or decision models. OVM is closely related to decision modeling where each variation point is a decision. CVL's variability specification tree is an enhanced feature model with cardinalities [12]. Delta modeling accepts any form of feature or decision model.

Featherweight VML employs feature trees and allows abstract features with no implementation [17]. Also, by employing a constraint language we can define any kind of dependencies between features.

The realization of features by artifacts is done in multiple ways. OVM uses an annotative approach to mark which artifacts are implementing specific decisions. Delta modeling uses a transformational approach to add, remove and modify artifacts from the model. A delta module's effects can span over the implementation of multiple features so it is not restricted by the structure of a feature tree. CVL variation points, especially the fragment substitution, can define complex transformations. However, they are directly bound to variability specifications so they are constrained by the tree structure. Featherweight VML uses fragment substitutions exclusively. The other CVL variation points, delta modules and of the OVM annotative technique can be reproduced by employing fragment substitutions.

Product derivation requires a clear understanding of how to execute a variability model given a specific configuration. CVL defines how each kind of variation point is executed. The variation points are partially ordered by the resolution tree structure. However, execution is not confluent as two variation points at the same level can have contradictory effects resulting in different variants depending on the order. OVM uses a projection on the model artifacts referenced by the selected variants. Delta Modeling executes each delta module by adding, modifying and removing elements as specified by the modules. The modules also specify a partial order using special clauses. The execution can be made confluent by adding conflict resolving deltas for any pair of conflicting deltas [18].

Orthogonality of variability modeling is the degree to which variability is modeled as a separate concern [19]. CVL defines a clear distinction between feature modeling (via a variability specification tree), and the model transformations over artifacts (via variation points). The variability model is completely separate from the artifacts. OVM design is based on orthogonality. The artifacts can be anything from requirements to model elements or code fragments.

Delta Modeling can be applied to any language, textual and graphical alike. Delta modules can use references to artifacts in a separate model to specify what is added, removed and modified. Featherweight VML borrows the layered architecture of CVL as it is general enough to be used with OVM and Delta Modeling.

3. Verifying Execution Correctness through Translation Validation

3.1. Correctness Properties

Product variant derivation is a process which takes as input *a feature model* over *a base system model* and *a configuration of features*. The output of the derivation is also *a system model* which contains only some of the artifacts of the input base model as specified by the configuration.

The correctness of a tool that implements variant derivation involves several facets (similar to the correctness of a generic model transformation tool [20]):

- termination of the transformation algorithm;
- confluence of the transformation steps (e.g. rule applications, fragment substitutions);
- obtaining the output model prescribed by the input.

The *termination* of the algorithm is essential to obtaining an output model. However, from a functional safety perspective, non-termination of a variant derivation tool is merely a nuisance for the developers of the product, but can't cause unsafe situations for the users of the product (since it is never created). Thus, functional safety standards do not consider termination as a requirement for tool qualification.

Non-*confluent* semantics introduces two problems. First, the product derivation process might be non-deterministic, leading to the possibility of obtaining different outputs on separate executions with the same input. This lowers the efficiency of testing in establishing functional safety. Second, the product derivation tool might be very sensitive to input such that, in spite of a deterministic implementation, the non-confluence is resolved in ad hoc way, not clearly related to input. This makes iterative quality improvement difficult, because small changes to the input model may have unexpected effects on the output.

There are two ways of making a transformation language semantics confluent: either by restricting the legal inputs so that there is no ambiguity (and providing warnings for the users in the case of illegal inputs) or by weakening the semantics in such a way that the non-confluence is hidden (for instance merging all possible outputs). The latter approach can't be used in product derivation, where a single output is needed. Therefore, we follow the former method, ensuring confluence by imposing constraints on the input model, thus eliminating all ambiguity of the specifications and providing a deterministic way of producing a single output, or an error message otherwise.

For product derivation, the confluence of the transformation algorithm can be proven independently of any execution and it holds for all the inputs. It just remains to be checked that the individual output models are consistent with the abstract specification used in this proof.

Additionally to confluence, we also need to verify that the output is as prescribed by the input,which is not a trivial task. This does not imply that the output is correct with respect to any external metamodel. It simply requires that all the specifications of the input model have been respected when producing the output. The problem increases with the complexity of the algorithm, but also with the size of the input. Instead of attempting to prove that the property holds for all executions, we use translation validation to verify the property on each individual resulting output.

Our goal is to verify these correctness properties which attest to the trustworthiness of the derivation tool. They refer to the correctness of the tool itself as opposed to the correctness of the output with respect to other external criteria. To illustrate the difference, we will not verify:

- the conformance of the output model to a language syntax or metamodel;
- the correctness of the output by checking that satisfies any safety properties;
- the introduction of errors resulting from feature interactions.

All the latter properties have been extensively researched, while the correctness of the tool has been largely assumed.

3.2. Translation Validation

Numerous tools exist for variability modeling with both commercial and academic implementations (e.g. pure::variants[3], BigLever's Gears[4], CVL[5], FeatureMapper[6], Clafer[7]). These tools are often part of sophisticated integrated development environments. The variant derivation process may use many complex software components, making it very challenging to develop a formal proof of the entire implementation. Even if such a proof could be developed, it would have to be updated whenever the implementation is updated due to bug fixes or addition of new features.

Translation validation [11] is a more pragmatic alternative to verifying translator tools (compilers, code generators). Instead of verifying that the tool *always* produces correct results, we can verify that each result is correct and conforming to the input only for the inputs on which the tool is actually run. The original translation validation method [11] relies on several ingredients:

[3] http://www.pure-systems.com/pure_variants.49.0.html
[4] http://www.biglever.com/solution/product.html
[5] http://www.omgwiki.org/variability/doku.php
[6] http://featuremapper.org/
[7] http://www.clafer.org/

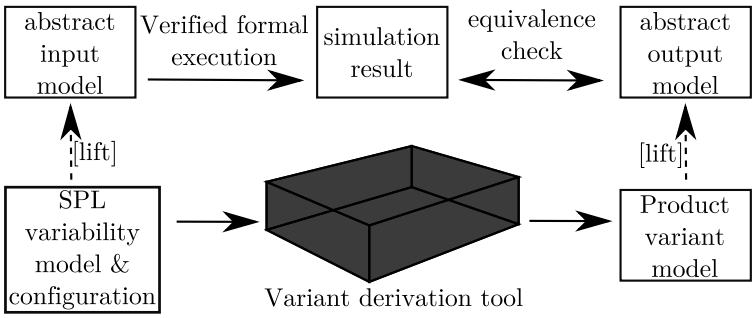

Figure 4: The translation validation process.

1. A common semantic framework for the representation of the source code and the generated target code.
2. A formalization of the notion of *correct implementation* as a refinement relation, based on the common semantic framework.
3. A proof method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source.
4. Automation of the proof method, to be carried out by an analyzer which, if successful, will also generate a proof script.
5. A rudimentary proof checker that examines the proof script produced by the analyzer and provides the last confirmation for the correctness of the translation.

We adapt the method to SPL variant derivation as illustrated in Fig. 4. The variant derivation tool is considered a black box. It can be an implementation of any variability modeling language. Its input is a variability model accompanied by a particular configuration. Its output is the product variant model.

For the original translation validation method, a *common semantic framework* for the representation of the input and output meant that a new semantic framework would be needed for each different language specification. By using Featherweight VML as the common semantic framework we can reuse the same setup for validating any tool for any language that can be abstracted to Featherweight VML. In Fig. 4 the abstraction is represented by the [lift]ing arrows.

We formalize the notion of *correct implementation* by providing formal semantics for Featherweight VML. We simulate the variant derivation via a formal execution of the abstract input model in Coq. The simulation result is then compared to the abstraction of the actual derivation result. If the two results are equivalent (isomorphic models) then we can say that the product variant model conforms to the input configuration. All this can be done by implementing a simpler tool than the actual derivation tool.

The advantages of simulating the derivation on an abstract model are multiple. The simulation tool can be reused to validate various production tools; the same simulation tool could work for translation validation of both Featherweight VML and CVL. While the actual tools depend on external libraries, the simulation tool can be implemented stand-alone, without dependencies to unverifiable compoents. Since the simulation is performed on abstractions of the models, the simulation tool can cave a smaller source code and is easier to verify. While most production tools are written in imperative languages, the simulation tool can be written using declarative or functional languages for which it is easier to write proofs of correctness.

We have implemented a simulation tool as a proof of concept (Sec. 5). We used Coq[8], an interactive theorem proving system implemented on top of a functional language. We implemented Featherweight VML as total functions which are shown to terminate. Then we developed theorems and proved that the simulation executes correctly and is confluent, i.e. the semantics of Featherweight VML is deterministic. Finally we can verify that the result of executing the abstract input model is equivalent to the abstraction of the output model. This approach has the advantage that we do not need to produce proof objects every time we perform the validation. The simulation is verified only once and can be reused on any lifted model. In the following sections we present the different elements of the setup in detail.

4. Featherweight VML

Featherweight VML is designed as a core variability modeling language. It is intended to provide a common framework to which languages such as CVL, Delta Modeling and OVM can be reduced. It is a formally defined language meant to offer a simple, unambiguous view of variability models and variant derivation. Our aim is to use Featherweight VML as a foundation for applying translation validation to variant derivation tools for actual languages (e.g. CVL, Delta Modeling, OVM).

In order to cover the entire variant derivation process, Featherweight VML must provide a syntax for abstracting base system models, a syntax for representing the variability specification and a semantics for variant derivation. While other formal specifications for graph-like model representation and transformation exist [21, 22, 23, 24, 25], using a specialized language for variant derivation has advantages not only from usability point of view, but also from a formal point of view: it is much easier to do verification (hereunder translation validation) for a tool that implements a concise formal language than for a big transformation language (e.g. ATL [24]).

[8] http://coq.inria.fr/

4.1. Abstract Model Representation

Featherweight VML is designed to specify variability in models defined using MOF-based metamodels, consisting of objects and relationships between them. We represent models as multi-graphs of attribute-less, untyped objects connected by directed links. We write $\mathbb{O}$ (respectively $\mathbb{L}$) to denote the infinite universe of all objects (resp. links). Both objects and links are discrete identifiable entities. The links are equipped with endpoint mappings indicating source and target objects: $\mathsf{src}\ l$ and $\mathsf{tgt}\ l$, both total functions of type $\mathbb{L} \to \mathbb{O}$. We assume that the universe of links is complete, in the sense that it contains infinitely many links with unique identities between any two objects in $\mathbb{O}$.

Definition 1. A *model* m is a pair of sets of finitely many objects and finitely many links, $m = (m_{\mathsf{Obj}}, m_{\mathsf{Lnk}}), m_{\mathsf{Obj}} \subseteq \mathbb{O}, m_{\mathsf{Lnk}} \subseteq \mathbb{L}$. A *model fragment* is a subset of objects and links of a model, so syntactically it is also a pair $f = (f_{\mathsf{Obj}}, f_{\mathsf{Lnk}})$.

While models and model fragments are syntactically identical, semantically they are different. We use the term *model* to refer to complete systems (i.e. the base model and the product variant). *Model fragments* represent components or incomplete pieces of models. We use *model fragments* as interchangeable units in the definition of fragment substitutions.

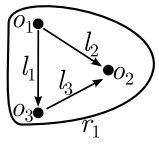

Figure 5: A fragment

We say that a model (or a fragment) m is *closed under links* (or simply *closed*) if for each link $l \in m_{\mathsf{Lnk}}$ its endpoints are contained in the model, so $\mathsf{src}\, l, \mathsf{tgt}\, l \in m_{\mathsf{Obj}}$.

Figure 5 illustrates a closed model fragment $r_1 = (\{o_1, o_2, o_3\}, \{l_1, l_2, l_3\})$. For the remainder of the paper we lift set operators to fragment operators, e.g. $f_1 \dot{\subseteq} f_2$ means $f_{1\mathsf{Obj}} \subseteq f_{2\mathsf{Obj}} \wedge f_{1\mathsf{Lnk}} \subseteq f_{2\mathsf{Lnk}}$.

4.2. The Fragment Substitution Variation Point

We introduce the formal definition of Featherweight VML in two steps: first we explain the execution of fragment substitutions, then we define the entire variability model relating feature models and fragment substitutions.

Syntax of the fragment substitution. Fragment r_1 introduced in Fig. 5 represents a component that can be customized by replacing o_2 with a new object, o_4. In Fig. 6a we define a *placement* fragment, p_1 (enclosed by a dashed line), containing the elements that must be removed from r_1. We also define a new *replacement* fragment, r_2 (enclosed by a solid

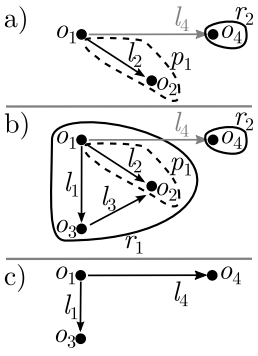

Figure 6: a) A fragment substitution.
b) Fragment interaction.
c) The execution result.

line), containing the elements that must be added. Finally, we create a new link, l_4 (represented by a gray arrow), that binds r_2 to the rest of the model. The placement and replacement fragments, p_1 and r_2, together with the new link, l_4, constitute a fragment substitution. Figure 6b shows how the fragment substitution interacts with r_1. After execution we obtain the result shown in Fig. 6c. The link l_3 was removed even though it was not part of the placement fragment, in order to avoid dangling links.

Definition 2. A *fragment substitution* fs is a triple (p, r, b) where p is a placement fragment containing all the elements that must be removed, r is the replacement fragment and b is a set of new links called a *binding*. The placement and replacement fragments are disjoint, $p \dot{\cap} r = (\emptyset, \emptyset)$.

Most variability modeling languages mark a model fragment to be copied by default and form the common base of any product variant (e.g. the core module in Delta Modeling). In order to keep the number of concepts low, in Featherweight VML we use fragment substitutions to represent both the common base and the subsequent changes applied to it. The example in Fig. 7a,b,c,d illustrates a set of fragment substitutions. We assume that we start from an empty model and fs_1 has only a replacement fragment which introduces the common base. The remaining substitutions perform further customization: fs_2 and fs_3 are removing the elements of p_1 and attach two other fragments, r_2 and r_3. The substitution fs_4 attaches a new fragment so its binding links have endpoints in r_3. Figure 7e represents the interactions between all fragment substitutions in a single base model. Figure 7f represents the substitutions with the Featherweight VML abstract syntax and Fig. 7g shows the final result.

The placement and replacement fragments are interchangeable units that can be defined independently of any fragment substitution. They can be seen as templates that can be reused. In Fig. 7f the placement fragment p_1 is referenced in both fs_2 and fs_3. Similarly, a replacement fragment can be reused in multiple fragment substitutions. Binding links, on the other hand, are dependent of a particular fragment substitution. They allow the reuse of placement and replacement fragments by changing only the way they connect. This is optimal in the cases where the fragments are large in the number of objects and links, but they connect through a small number of links defined as bindings. In the case when a fragment substitution is required to link objects that have been introduced by previous substitutions, the replacement fragment may be empty and the operation can be performed through binding links exclusively.

We require that for any fragment substitution $fs = (p, r, b)$, the binding links are not incident with placement objects, $\forall l \in b(\mathsf{src}\, l \cup \mathsf{tgt}\, l) \cap p_{\mathsf{Obj}} = \emptyset$. All such links would be removed as dangling since their endpoints belonging to a placement would be removed. Binding links can only be incident with objects from the replacement fragment and boundary objects, defined as follows:

Definition 3. The *boundary* of a fragment substitution $fs = (p, r, b)$ is the set of all endpoints of binding links that are not part of the replacement fragment: $\mathsf{boundary}\, fs = \{o \mid o = \mathsf{src}\, l \vee o = \mathsf{tgt}\, l, l \in b\} \setminus r_{\mathsf{Obj}}$.

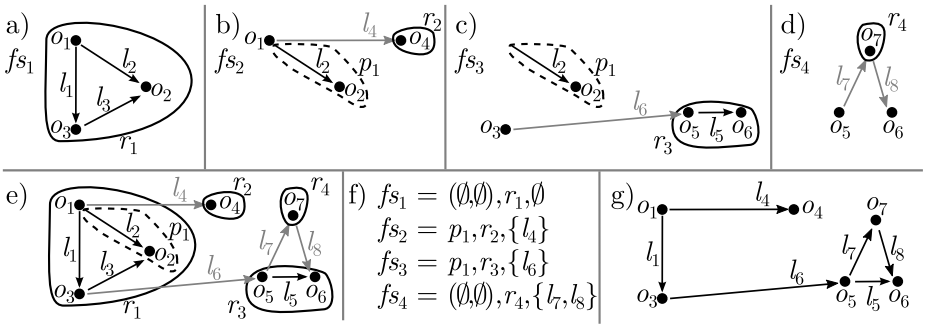

Figure 7: a,b,c,d) A set of fragment substitutions. e) Interactions between fragment substitutions. f) Syntactic representation. g) The execution result.

In Fig. 7 we have $\mathsf{boundary}\, fs_2 = \{o_1\}$, $\mathsf{boundary}\, fs_3 = \{o_3\}$ and $\mathsf{boundary}\, fs_4 = \{o_5, o_6\}$. In Sec. 4.3 we will need to identify all artifacts that a fragment substitution affects outside of its own replacement fragment. These are the artifacts in the placement fragment and the boundary objects used by the binding links.

Definition 4. Given a fragment substitution $fs = (p, r, b)$, the *closure* of the placement fragment p, written $\lceil p \rceil_{fs}$, is defined as all objects of p plus the boundary of the fragment substitution; the set of links remains unchanged: $\lceil p \rceil_{fs} = (p_{\mathsf{Obj}} \cup \mathsf{boundary}\, fs, p_{\mathsf{Lnk}})$.

In Fig. 7, $\lceil p \rceil_{fs_2} = (\{o_1, o_2\}, \{l_2\})$, $\lceil p \rceil_{fs_3} = (\{o_2, o_3\}, \{l_2\})$ and $\lceil p \rceil_{fs_4} = (\{o_5, o_6\}, \emptyset)$. Substitutions fs_2 and fs_3 have different placement closures even if they refer to the same placement fragment. This is because the binding links differ.

Execution semantics of the fragment substitution. The example in Fig.7 gave the intuition of the fragment substitution execution process. Instead of performing in-place changes to the base model, we propose a copying semantics, meaning that we decide for each object/link whether it should be part of the product variant and we copy only those for which we decide positively.

Given a set of fragment substitutions, Fs, we will copy all replacement fragments and all binding links. However, we know that what is contained by placement fragments should be removed and replaced so we will not copy these elements. We will not copy links that are incident with placement fragments either. The result $[\![Fs]\!]$ of executing a set of fragment substitutions Fs is called a product variant model; it is a pair of sets of objects/links. The following rules precisely describe which objects and links are copied in $[\![Fs]\!]$:

$$\frac{o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \quad o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}})}{o \in [\![Fs]\!]_{\mathsf{Obj}}}\ (\textsc{obj-copy}) \qquad \frac{l \in (\bigcup_{(\_,r,b)\in Fs} r_{\mathsf{Lnk}} \cup b) \quad l \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Lnk}}) \quad \mathsf{src}\, l, \mathsf{tgt}\, l \notin \bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}}}{l \in [\![Fs]\!]_{\mathsf{Lnk}}}\ (\textsc{lnk-copy})$$

The OBJ-COPY rule says that any object contained in a replacement fragment of a fragment substitution in Fs will be copied as long as it is not contained in any placement fragment. The LNK-COPY rule says that any link that is contained in a replacement fragment or in a binding set of a fragment substitution in Fs will be copied as long as the link or its endpoints are not contained in any placement fragment. The rules are applied exhaustively for all objects and links in all fragments and bindings in the set of fragment substitutions. The complete input model is illustrated in Fig. 7e. Even though individual fragments do not have to be closed under links (see page 14), the complete input model may be closed. Lemma 1 ensures that applying the rules to a closed input model results in a product variant model without any dangling links.

Lemma 1. *Given a set of fragment substitutions Fs such that the union of all placement fragments, replacement fragments and bindings is a closed model, the product variant model* $[\![Fs]\!]$ *is also closed under links.*

PROOF. (Sketch) By assumption, the union of all objects and links is a closed graph, so for every link that might be copied, the graph also contains its endpoints. Then we notice that the premise of (LNK-COPY) is that neither the source or the target of the link being copied are contained in a placement fragment. Thus it is guaranteed that for any link that is being copied, both link ends will also be copied.

Lemma 2. *Given a set of fragment substitutions, there exists a unique product variant model created by the above rules.*

The lemma holds by construction: objects and links are deterministically selected from a finite set. It follows from the Lemma 2 that the execution of fragment substitution sets is order independent (in other words the semantics is *confluent*), which opens for various implementation strategies.

4.3. The Variability Model

We have shown how to execute a set of fragment substitutions, Fs, to obtain a product variant model. In a normal scenario, we would like Fs to describe multiple variants and to be able to select only those fragment substitutions that describe a specific product variant before executing them. We would also like to be able to execute a fragment substitution multiple times and to use a configuration to specify how many copies of the replacement fragment to include in the product variant.

Figure 8a illustrates a variability model where each fragment substitution, $fs_{1..4}$, is mapped to a feature, $ft_{1..4}$, from a feature tree. Each feature displays a cardinality constraint for how many instances are allowed for that feature under a single parent. In Fig. 8b the features are instantiated in a configuration tree. The root feature has one root instance, feature ft_2 is not instantiated and ft_3 is instantiated twice, meaning that its fragment substitutions should be executed twice. Feature ft_4 is only instantiated as a child of i_2.

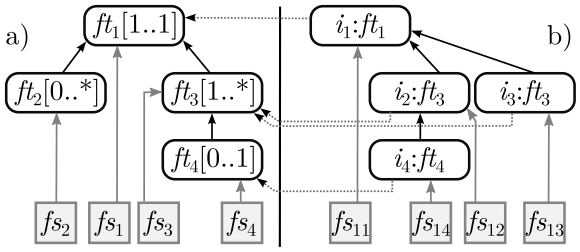

Figure 8: a) A variability model. b) A configuration and a flattened set of fragment substitutions.

Section 4.2 does not handle multiple execution of fragment substitutions. Instead we will show how to flatten the model and the chosen configuration in a set of fragment substitutions that contains as many copies of each fragment substitution as there are instances of its feature. Flattening the model in our example would result in a set containing two copies of fs_3, but no copies for fs_2.

Syntax of the variability model

A feature model defines all characteristics that can be activated in a product variant. Some characteristics may occur multiple times in a product variant (e.g. the number of USB ports on a computer). For this reason, a *feature* in Featherweight VML is similar to a type that can be instantiated multiple times in the product variant so our features have cardinality [12].

Definition 5. A *feature model* is a rooted directed tree of features, $Fm = (Ft, ft_0, \mathsf{parent})$, where Ft is a set of features, $\mathsf{parent} \subseteq Ft \times Ft$ is a connected acyclic parent relation with no sharing (a tree), and $ft_0 \in Ft$ is the root of the tree. We write $\mathsf{parent}\, ft_2 = ft_1$, if feature ft_1 is a parent node of ft_2 in Fm.

Each feature ft has an associated cardinality constraint $\mathsf{card}\, ft = (\mathsf{min}\, ft, \mathsf{max}\, ft)$, where $\mathsf{min}\, ft, \mathsf{max}\, ft \in \mathbb{N} \cup \{*\}$, $\mathsf{min}\, ft \leq \mathsf{max}\, ft$ (the symbol $*$ is considered greater than any natural).

A set of fragment substitutions and the feature model that controls which combinations of fragment substitutions can be executed together constitute a complete variability model.

Definition 6. A *variability model* is a triple, $(Fs, Fm, \mathsf{mapping})$, where Fs is a set of fragment substitutions, $Fm = (Ft, ft_0, \mathsf{parent})$ is a feature model and $\mathsf{mapping} : Fs \to Fm$ maps each fragment substitution to a feature.

A configuration represents a combination of features that are active in a product variant.

Definition 7. Given a feature model $Fm = (Ft, ft_0, \mathsf{parent})$, a *configuration* is a rooted tree $Cfg = (I, i_0, \mathsf{parent}, \mathsf{ty})$, where I is a finite set of *instances*, $i_0 \in I$ is the root of the tree, $\mathsf{parent} \subseteq I \times I$ is a connected acyclic parent relation with no sharing (a tree). The typing mapping $\mathsf{ty} : I \to Fm$ maps every instance to its feature, in a manner preserving the parent relations:

i. The root instance is typed by the root feature: $\mathsf{ty}\, i_0 = ft_0$,

ii. The children of an instance are typed by children of its type: for instances i, j, if $\mathsf{parent}\, i = j$ then $\mathsf{parent}\,(\mathsf{ty}\, i) = \mathsf{ty}\, j$.

iii. The feature cardinality constraints are satisfied, so for each instance $j \in I$ and feature $ft \in Ft$, if $\mathsf{parent}\, ft = \mathsf{ty}\, j$ then

$$\mathsf{min}ft \leq |\{i \in I \mid \mathsf{parent}\, i = j \text{ and } \mathsf{ty}\, i = ft\}| \leq \mathsf{max}ft$$

Before moving on to the execution semantics we give a set of well-formedness constraints that guarantee that the flattening of variability models produces unique sets of fragment substitutions that can be executed with the rules introduced in Sec. 4.2.

C 1. *The mapping of fragment substitutions to features is injective. Any two fragment substitutions, $fs_i = (p_i, r_i, b_i)$ and $fs_j = (p_j, r_j, b_j)$, that should be mapped to the same feature can be merged into a single fragment substitution, $fs_n = (p_i \dot{\cup} p_j, r_i \dot{\cup} r_j, b_i \cup b_j)$.*

Constraint 1 helps simplifying the following constraints and the semantics. It does not limit the expressive power of Featherweight VML. If fs_i and fs_j should anyway be mapped to the same feature then they should be executed together for each instance of that feature. Thus, requiring that they should be combined into one fragment substitution does not change their effect.

It is not required that every feature has a substitution mapped to it. The inverse $\mathsf{mapping}^{-1} : Ft \to [Fs \cup \{\bot\}]$ returns the fragment substitution mapped to a feature or $\bot$ if such a fragment substitution does not exist.

C 2. *All replacement fragments are closed under links. This constraint enforces that for any link cloned during flattening, its endpoints are also cloned and all the clones will be consistent with the original fragment.*

Figure 9 illustrates the replacement fragment problem fixed by constraint 2. Assume we have two replacement fragments r_1 and r_2 such that a link form r_2 has an endpoint in r_1. Each fragment is used in a fragment substitution and each substitution is mapped to a different feature. If we instantiate r_1 three

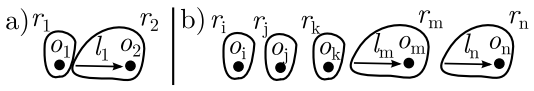

Figure 9: The replacement fragment problem.

times—resulting in fragments r_i, r_j and r_k being inserted in the product variant—and we instantiate r_2 two times—resulting in fragments r_m and r_n—then there is no clear intuition about which of the new objects should be used as endpoints for the new links. In fact, we could even instantiate the links, but not their endpoints.

When a fragment substitution fs_i is instantiated, product line developers intuitively assume that the artifacts of the placement fragment (provided that the fragment is not empty) have already been introduced in the output by the instantiation of another fragment substitution, fs_j. The execution order of these two fragment substitutions is influencing the confluence of the derivation process because fs_i removes and fs_j adds the same artifacts. Featherweight VML is addressing the confluence issue by enforcing constraints on the input variability models.

We recall that the closure of the placement fragment $\lceil p \rceil_{fs_i}$ of a fragment substitution $fs_i = (p_i, r_i, b_i)$ is composed of (i) all the artifacts that will be removed as being part of the placement fragment p_i extended with (ii) all the objects that are endpoints of binding links, but are not newly added by the replacement fragment r_i: $\lceil p \rceil_{fs_i} = (p_{i\mathsf{Obj}} \cup \mathsf{boundary}\, fs_i, p_{i\mathsf{Lnk}})$. More concisely, the placement closure of a fragment substitution is composed of all the artifacts that are removed or otherwise affected, but are now newly added by the fragment substitution itself. If the placement closure $\lceil p \rceil_{fs_i}$ is not empty and there exists one and only one other fragment substitution $fs_j = (p_j, r_j, b_j)$ such that the artifacts of $\lceil p \rceil_{fs_i}$ are added by fs_j (formally $\lceil p \rceil_{fs_i} \dot{\subseteq} r_j$) we say that fs_i *applies to* fs_j and we write $fs_i \sqsubset fs_j$.

C 3. *All artifacts that are removed or affected in any way by any fragment substitution, must be added by a different fragment substitution. For any fragment substitution $fs_i \in Fs$ for which the placement closure is not empty, $\lceil p \rceil_{fs_i} \dot{\neq} (\emptyset, \emptyset)$, there must exist another fragment substitution $fs_j \in Fs$ such that fs_i applies to fs_j, $fs_i \sqsubset fs_j$.*

Constraint 3 does not in its own enforce the confluence of the semantics. We also need to enforce that if fragment substitution fs_i applies to fs_j, then fs_j is executed first. This is to guarantee that the artifacts are added before removing them or binding to them. Furthermore, in the case that multiple clones are required for both fragment substitutions, the way in which the clones are created must be consistent with the structure of the feature model.

C 4. *The structure enforced by the application, $\sqsubset$, of fragment substitutions is consistent with the feature model: if $fs_i \sqsubset fs_j$ then $\mathsf{mapping}\, fs_j \in \mathsf{parent}^*(\mathsf{mapping}\, fs_i)$, so if one fragment substitution applies to another, then it is mapped to a feature in the subtree rooted by the feature of the other. Function parent^* is the reflexive transitive closure of parent.*

All constraints must be verified against the input variability model to ensure that an output can be derived deterministically from the variability specification. Once the variability model is flattened, the cloning and renaming of fragment

substitutions and artifacts eliminates the dependency on the structure of the feature model. After flattening, the execution order of the fragment substitutions does not affect the output product variant model.

Execution semantics of the variability model

In Fig. 10 we recall the fragment substitutions of Fig. 8. On the left side we have the detailed contents of the initial four fragment substitutions. On the right side we have the flattened set. The configuration does not contain an instance for ft_2 so fs_2 is not copied. The substitution fs_3 must be executed two times—once for the instance i_2 and once for i_3. Since the semantics presented in Sec. 4.2 only execute each substitution once, we flatten the model by computing how many times each fragment substitution should be executed and cloning it the appropriate amount of times (carefully updating references).

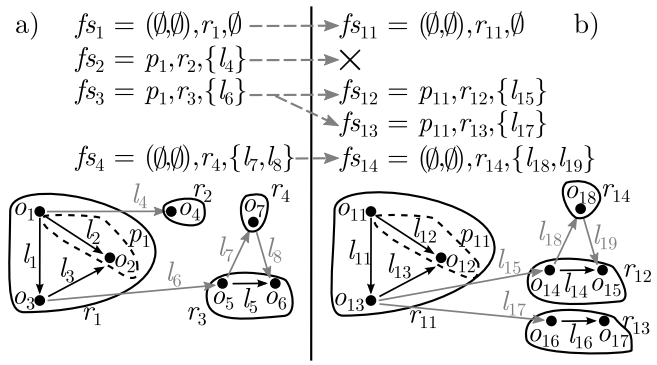

Figure 10: Illustration of the flattening process: a) before, b) after

The semantics is presented as follows: first we introduce functions for copying and renaming basic entities—objects and links. Second, we lift these functions to sets of objects/links, to base model fragments and to fragment substitutions. Third, we explain the flattening of variability models and configurations. We conclude the semantics with a theorem of confluence.

The renaming of objects and links is needed because multiple clones of the same artifact can occur in the same product variant. In the case of a variability model where each feature can only be instantiated once, the renaming is not necessary.

Preliminaries: Copying and renaming basic entities.

Given a variability model, $(Fs, Fm, \mathsf{mapping})$, we use the sets O and L to reference all artifacts contained in this model, $O = \bigcup_{(p,r,b)\in Fs} [p_{\mathsf{Obj}} \cup r_{\mathsf{Obj}}]$ and $L = \bigcup_{(p,r,b)\in Fs} [p_{\mathsf{Lnk}} \cup r_{\mathsf{Lnk}} \cup b]$.

Given a configuration $Cfg = (I, i_0, \mathsf{parent}, \mathsf{ty})$ we use the set I of instances as an index for renaming artifacts. Since the product variant model may end up containing several copies of the same artifacts, we will need to create fresh objects and links, and then be able to refer to them unambiguously. We model this using two injective functions $\mathsf{new\text{-}obj}$ and $\mathsf{new\text{-}lnk}$ that create new objects/links

for any given feature instance.

$$\mathsf{new\text{-}obj} : I \times O \to \mathbb{O} \setminus O \qquad \mathsf{new\text{-}lnk} : (I \times I) \times L \to \mathbb{L} \setminus L$$

We write the first argument in all renaming functions as an index to make the notation more lightweight. Intuitively, the first argument represents an ordinal index of the copy, whereas the second argument is the entity being copied.

We require that the two functions map to an isomorphic graph structure, so they are injective and for every pair of feature instances i, j (possibly but not necessarily, $i = j$) and any link l we have that: $\mathsf{src}\,(\mathsf{new\text{-}lnk}_{i,j} l) = \mathsf{new\text{-}obj}_i(\mathsf{src}\, l)$ and $\mathsf{tgt}\,(\mathsf{new\text{-}lnk}_{i,j} l) = \mathsf{new\text{-}obj}_j(\mathsf{tgt}\, l)$.

For every instance-object pair we get a different new object, which was not in O. Similarly, for every instance pair (i, j) and a link we get a link, which was not in L, connecting copies of the objects related with $\mathsf{new\text{-}obj}_i$ and $\mathsf{new\text{-}obj}_j$.

We lift the two functions to rename (create) entire sets of objects and links:

$$\mathsf{new\text{-}Obj} : I \times 2^O \to 2^{\mathbb{O}\setminus O}, \text{ where } \mathsf{new\text{-}Obj}_i O' = \{\mathsf{new\text{-}obj}_i o \mid o \in O'\} \text{ and}$$
$$\mathsf{new\text{-}Lnk} : (I \times I) \times 2^L \to 2^{\mathbb{L}\setminus L}, \text{ where } \mathsf{new\text{-}Lnk}_{i,j} L' = \{\mathsf{new\text{-}lnk}_{i,j} l \mid l \in L'\}.$$

Such renaming functions always exist due to our assumption that the universes of objects and links are complete and infinite and we can always obtain a new link between any two objects.

Copying fragments and bindings. We will now explain how to copy a fragment substitution such that all its clones (each clone implementing a different instance) are independent of each other. We lift the simple renaming functions shown above to fragments:

$$\mathsf{new\text{-}frg}_i(O', L') = (\mathsf{new\text{-}Obj}_i O', \mathsf{new\text{-}Lnk}_{i,i} L').$$

In our example we copy the fragment r_3 for the instances i_2 and i_3:
$\mathsf{new\text{-}frg}_2(\{o_5, o_6\}, \{l_5\}) = (\mathsf{new\text{-}Obj}_2\{o_5, o_6\}, \mathsf{new\text{-}Lnk}_{2,2}\{l_5\}) = (\{o_{14}, o_{15}\}, \{l_{14}\})$,
$\mathsf{new\text{-}frg}_3(\{o_5, o_6\}, \{l_5\}) = (\mathsf{new\text{-}Obj}_3\{o_5, o_6\}, \mathsf{new\text{-}Lnk}_{3,3}\{l_5\}) = (\{o_{16}, o_{17}\}, \{l_{16}\})$.

Renaming bindings is more complex—the endpoints may be renamed differently, according to which fragment they belong to. We formalize binding renaming to take as parameter two disjoint sets of objects. We apply i-renaming if an endpoint is in the first set, and j-renaming if the endpoint is in the other set:

$$\mathsf{new\text{-}bdg}_{i,j}(O_1, O_2, L) = \{\mathsf{new\text{-}lnk}_{\mathsf{ns}(\mathsf{src}\, l),\mathsf{ns}(\mathsf{tgt}\, l)} l \mid l \in L\},$$

where ns is a function mapping objects to name spaces (instances), depending on which replacement they belong to; $\mathsf{ns}\, o$ returns i if $o \in O_1$ and it returns j if $o \in O_2$. In our example we want to copy the binding links l_7 and l_8. The ns function allows us to copy the source of l_7 and target of l_8 with the appropriate instance i_2: $\mathsf{new\text{-}bdg}_{4,2}(\{o_7\}, \{o_5, o_6\}, \{l_7, l_8\}) = \{\mathsf{new\text{-}lnk}_{2,4} l_7, \mathsf{new\text{-}lnk}_{4,2} l_8\} = \{l_{18}, l_{19}\}$.

Finally, we lift the renaming functions to entire fragment substitutions:

$$\mathsf{new\text{-}fs}_{i,j}(p, r, b)\, O_j = \left(\mathsf{new\text{-}frg}_j p, \mathsf{new\text{-}frg}_i r, \mathsf{new\text{-}bdg}_{i,j}\,(r_{\mathsf{Obj}}, O_j, b)\right).$$

Intuitively, if objects are in set O_j then they should be renamed using the j-indexed renaming functions. It they are in the replacement of the fragment substitution then the i-indexed renaming functions apply. The set O_j will be provided in the semantics according to the context, and it should always be disjoint from objects of the replacement r_{Obj}.

In our example copying fs_3 for i_2 is done by copying p_1 for i_1, r_3 for i_2 and the binding link has its source copied for i_1 and its target for i_2:
$\mathsf{new\text{-}fs}_{2,1}(p_1, r_3, \{l_6\})\, r_1 = \left(\mathsf{new\text{-}frg}_1 p_1, \mathsf{new\text{-}frg}_2 r_3, \mathsf{new\text{-}bdg}_{2,1}\,(\{o_5, o_6\}, r_1, \{l_6\})\right).$

Flattening variability models and configurations. By constraint 1 we know that there can be only one fragment substitution mapped to any feature, but it is not required that every feature has a substitution mapped to it. Each feature can be instantiated multiple times in which case the fragment substitution mapped to it (if it exists) is executed multiple times (once per instance). We compute how many times each substitution should be executed and clone it the appropriate amount of times (carefully updating references). This will produce a flat set of fragment substitutions that can be executed using the rules of Sec. 4.2.

The flattening of a variability model M with respect to a configuration Cfg is a set of fragment substitutions, denoted below as $[\![M, Cfg]\!]$. Flattening moves all the necessary information from the feature model and from the fragment substitutions to a new set of fragment substitutions. After this, the features and their instances can be disregarded.

Given a variability model $M = (Fs, Fm, \mathsf{mapping})$ and a configuration Cfg, $\mathsf{mapping}^{-1}(\mathsf{ty}\, i)$ returns the fragment substitution that has to be executed in the context of an instance i or $\bot$ if there is no such substitution.

There are three cases to consider when flattening the model. In the first case, instances of features that have no substitutions mapped to them are ignored by the semantics. In the second case, instances of features that have substitutions with empty placement closures such that they do not apply to any other substitution are copied with the following rule:

$$\frac{i \in Cfg \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, i) = fs_i \quad \lceil p \rceil_{fs_i} = (\emptyset, \emptyset)}{\mathsf{new\text{-}fs}_{i,\_}\; fs_i\; \emptyset \in [\![M, Cfg]\!]}\ (\textsc{copy-indep})$$

Since the placement fragment is empty and the binding links endpoints can only be objects of the replacement fragment itself, binding links can be appropriately cloned by using just the instance i, by $\mathsf{new\text{-}fs}$.

In the third case, instances of features that have substitutions which apply to other substitutions are copied with the following rule:

$$\frac{\begin{array}{c} i, j \in Cfg \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, i) = fs_i \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, j) = fs_j \quad fs_i \sqsubset fs_j \\ fs_i = (p_i, r_i, b_i) \quad fs_j = (\_, r_j, \_) \quad j \in \mathsf{parent}^{*} i \end{array}}{\mathsf{new\text{-}fs}_{i,j}\; fs_i\; r_j \in [\![M, Cfg]\!]}\ (\textsc{copy})$$

The intended meaning of COPY is that we copy the replacement fragment using the instance i, the placement with the instance j and the binding links with a combination of the two. We use r_j, the replacement fragment of fs_j to state that a binding link endpoint can either be in the r_i or r_j. By constraint 1 we know that for any pair of instances i and j, $\mathsf{mapping}^{-1}(\mathsf{ty}\, i)$ and $\mathsf{mapping}^{-1}(\mathsf{ty}\, j)$ are uniquely determined (if they exist), thus the rule can be applied deterministically.

In our example we know that $i_2, i_1 \in \mathit{Cfg}$, $\mathsf{mapping}^{-1}(\mathsf{ty}\, i_2) = fs_3$ and $\mathsf{mapping}^{-1}(\mathsf{ty}\, i_1) = fs_1$, $fs_3 \sqsubset fs_1$ and $i_1 \in \mathsf{parent}^* i_2$, therefore we copy fs_3 in the flattened set: $\mathsf{new\text{-}fs}_{2,1}(p_1, r_3, \{l_6\})\, r_1 \in [\![M, \mathit{Cfg}]\!]$.

Lemma 3. *For a well-formed variability model M and a valid configuration Cfg, the above rules define a unique well-formed set of fragment substitutions $[\![M, Cfg]\!]$.*

The well-formedness of the output follows from the isomorphism of all renaming operations (all functions are injective and preserve links)—all non-overlapping conditions of well-formedness are thus transferred from the input set of fragment substitutions.

Theorem 1. *Given a well-formed variability model M and a valid configuration Cfg the result of executing the model is unique, and given by $[\![M, Cfg]\!]$, and consequently the above formulation of the semantics is confluent.*

The well-formedness constraints (C 1,2,3,4) ensure that the flattening input set of fragment substitutions form a closed union of fragments. Lemma 3 ensures that the output of the flattening is a unique set of substitutions forming a closed union of fragments. Lemma 3 ensures that copying process results in a closed product variant model and Lemma 2 ensures that the result is unique regardless of the ordering of the input objects and links.

5. Translation validation for Featherweight VML

As proof of concept we implemented the translation validation mechanism for the execution of a fragment substitution set[9]. This covers the syntax and semantics presented in Sec. 4.2.

As we explained in Sec. 3, translation validation has several requirements. We present the common semantic framework for the input and output as the implementation of Featherweight VML in Coq in Sec. 5.1. A formalization of the notion of *correct implementation* and the simulation of product derivation in Sec. 5.2. An *automatic proof generator* and a *proof checker* are also required to perform the validation. These are fundamental features of Coq and we explain how we use them in Sec. 5.3. Then we describe Micro CVL, a variant derivation tool developed using the Eclipse Modeling Framework (EMF)[10] that works on

[9] https://github.com/afla/FeatherweightVML_Coq
[10] http://www.eclipse.org/modeling/emf/

Ecore models in Sec. 5.4. Finally we describe how to lift the input and output of Micro CVL to Coq abstractions and validate the derivation in Sec. 5.5.

5.1. Fragment substitution syntax in Coq

Listing 2 shows the core syntax of abstract models and fragment substitutions. In Featherweight VML objects and links are discrete identifiable entities. We use the predefined set of natural numbers (nat) as identifiers. To define new types we can either use the *Definition* keyword to rename existing types or the *Inductive* keyword to specify how new types can be created by composing existing ones. Line 1 defines objects to be simple identifiers without any additional properties. Line 2 defines a type for sets of objects by reusing the predefined parametrized type list. This definition does not enforce that ObjectSets are actually sets, as lists are ordered and can contain the same element multiple times. We will later define properties to enforce this. Line 4 defines the Link type. The constructor, link, takes a nat identifier and two Objects, representing the source and the target of a Link. Line 6 sets up a concise infix notation for links. Similarly a Graph is a pair of sets of objects and links (line 8). For the Model and Fragment types we simply reuse the abstract type Graph, reflecting the syntax defined in Sec. 4.1. The FragSubst type of fragment substitutions combines two fragments and a set of links (line 13), the *placement*, *replacement* and *binding* as in Def. 2.

```
1  Definition Object := nat.
2  Definition ObjectSet := list Object.
3
4  Inductive Link := link : nat -> Object -> Object -> Link.
5  Definition LinkSet := list Link.
6  Notation "id ¤ src -–> tgt" := (link id src tgt) (at level 66).
7
8  Inductive Graph := graph : ObjectSet -> LinkSet -> Graph.
9  Notation "gObj ** gLnk" := (graph gObj gLnk) (at level 64).
10 Definition Model := Graph.
11 Definition Fragment := Graph.
12
13 Inductive FragSubst :=
14   fragsubst : Fragment -> Fragment -> LinkSet -> FragSubst.
15 Notation "p ,. r ., b" := (fragsubst p r b) (at level 65).
16 Definition FragSubstSet := list FragSubst.
```

Listing 2: Abstract syntax of Featherweight VML in Coq.

Featherweight VML relies heavily on set theory. When we encoded Featherweight VML in Coq we also implemented the basic properties, relations and operations of set theory. All the properties and relations are decidable and Coq can automatically compute whether they hold for any concrete sets or set elements. For example, Listing 3 shows the SetContainsObject property which tests whether a set contains a particular Object. In Coq, properties are a special kind of inductive types. The constructor SetContainsObject_h specifies the base case when the Object is accessible as the head of the list, while the constructor

SetContainsObject_t specifies the inductive case when the Object is contained in the tail of the list.

In order to prove that an actual ObjectSet contains a specific Object, so that property SetContainsObject holds for them, we would have to manually build a proof object, establishing SetContainsObject, by repeatedly applying the two constructors. Instead, we demonstrate that the property is decidable and provide Coq with the means to automatically check the property for any input. Lines 6 and 7 define SetContainsObject_dec, a decidable type that holds both the fact that an ObjectSet contains a specific Object or not and the proof object. The notation on line 7 indicates a **left** constructor for the positive evaluation of the property and a **right** constructor for the negative evaluation.

At this point we can use Coq as an automatic proof generator. Lines 9 to 16 define setContainsObject, a theorem refined as a fixpoint that calculates the proof object. The fixpoint iterates through the ObjectSet and if it manages to build a complete proof object it returns the **left** constructor of the decidable type SetContainsObject_dec. Otherwise it calls the **right** constructor. Lines 17 to 21 represent the proof that the fixpoint is correct. Executing setContainsObject on an actual ObjectSet and an actual Object allows Coq to generate the proof for that particular case. This approach is much easier and scales very well compared to generating proofs for large models from outside Coq. It is also more reliable and can be performed automatically on any input. In a similar way we have proven that all our other properties are decidable and computable by Coq.

```
1  Inductive SetContainsObject : ObjectSet -> Object -> Prop :=
2    | SetContainsObject_h : forall o t, SetContainsObject (o::t) o
3    | SetContainsObject_t : forall o h t,
4                            SetContainsObject t o -> SetContainsObject (h::t) o.
5
6  Definition SetContainsObject_dec (s : ObjectSet) (o : Object) :=
7    { SetContainsObject s o } + { ~SetContainsObject s o }.
8
9  Definition setContainsObject: forall s o, SetContainsObject_dec s o.
10 refine (fix setContainsObject s o: SetContainsObject_dec s o :=
11   match s with
12   | [ ] => right _
13   | h::t => if eq_nat_dec h o
14             then left _
15             else if setContainsObject t o then left _ else right _
16   end).
17 Proof.
18   unfold not. intro. inversion H. rewrite _H. apply SetContainsObject_h.
19   apply SetContainsObject_t. apply _H0. unfold not. intro. inversion H.
20   apply _H. apply H0. unfold not in _H0. apply _H0. apply H3.
21 Defined.
```

Listing 3: A set membership property along with a function that computes the property for any ObjectSet and Object.

The equality property differs for most types. In Featherweight VML two

objects are equal if their identifiers are equal. The same is true for links, so in Coq we use the property LinkEqualId to test that 3¤6–>8 and 3¤1–>4 are equal and cannot be contained in the same LinkSet. However, the same link identifier can occur in multiple link sets, or model fragments, or fragment substitutions. We must check that the source and target objects are the same in all occurrences of the link. We call this property *link consistency*. We do this by defining a series of link consistency properties which can be checked for any two structures that contain links.

Sets of any type are equal if they contain the same elements. For graphs (i.e. models and fragments) equality is verified pair-wise on the object and link sets. Fragment substitutions are equal if the placement/replacement fragments and bindings are equal respectively.

5.2. Fragment substitution semantics in Coq

Before we could encode the two copying rules, OBJ-COPY and LNK-COPY (Sec. 4.2), we had to implement some helper functions that, given a set of fragment substitutions, extract the following sets: all placement objects, all placement links, all replacement objects/links, all binding links and all objects/links throughout the entire fragment substitution set. For example, Lst. 4 shows a function that computes the set of all placement objects from all placement fragments of the fragment substitution set fss. We then verify that it executes correctly by proving the theorem PlacementObjectsExecutes which states that for any fragment substitution (pObj**pLnk,.r.,b) contained by fss, it is implied that pObj will be a subset of the complete set of placement objects. Theorem PlacementObjectsOfEqualFss states that any two fragment substitution sets that are equal (i.e. contain the same elements but possibly in different orders) contain the same placement objects. The theorems have a double role: first, they are improving the quality of the translation validator by having extra checks of the execution and second, they are used in proving larger theorems. We provide similar theorems for all executable functions.

```
1  Fixpoint placementObjects (fss : FragSubstSet) : ObjectSet :=
2  match fss with
3    | [ ] => [ ]
4    | (pObj**pLnk,.r.,bdg)::t => objectSetUnion pObj (placementObjects t)
5  end.
6
7  Theorem PlacementObjectsExecutes : forall fss pObj pLnk r b,
8    SetContainsFragSubst fss (pObj**pLnk,.r.,b)
9    -> ObjectSubset pObj (placementObjects fss).
10
11 Theorem PlacementObjectsOfEqualFss : forall fss1 fss2,
12   FragSubstSetEqual fss1 fss2 ->
13   ObjectSetEqual(placementObjects fss1)(placementObjects fss2).
```

Listing 4: A function that computes all the placement objects from a set of fragment substitutions along with two theorems stating the correctness of this computation

Finally we have implemented the two copy rules. Listing 5 shows the implementation of OBJ-COPY. Given a set of objects obj and a set of fragment substitutions fss, objCopy will check if the head of obj is a replacement object in fss and in the same time is not a placement object in fss. If the conditions are met, then the head object is copied and the function is called recursively on the tail, otherwise the object is not copied and the tail is processed. The implementation of lnkCopy is analogous to objCopy.

$$\frac{o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \quad o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}})}{o \in [\![Fs]\!]_{\mathsf{Obj}}} \text{ (OBJ-COPY)}$$

```
1 Fixpoint objCopy (obj : ObjectSet) (fss : FragSubstSet) : ObjectSet :=
2 match obj with
3   | [ ] => [ ]
4   | h::t =>
5    if bSetContainsObject (replacementObjects fss) h
6      && !(bSetContainsObject (placementObjects fss) h)
7    then h::(objCopy t fss)
8    else (objCopy t fss)
9 end.
```

Listing 5: Implementing the OBJ-COPY inference rule as a function

The execution of a fragment substitution set, presented in Listing 6, is simply composing a variant model by applying objCopy to all objects in fss and lnkCopy to all links in fss.

```
1 Function executeFss (fss : FragSubstSet) : Model :=
2    (objCopy (allObjects fss) fss)**(lnkCopy (allLinks fss) fss).
```

Listing 6: Implementation of the execution function as the combined application of objCopy and lnkCopy

5.3. Proof of correctness and confluence

Independently, we also wanted to ensure that the Coq implementation of Featherweight VML is of good quality. Listing 7 shows theorem objCopyExecutes which states that the function objCopy correctly implements the copying rule OBJ-COPY. An interesting aspect of this theorem is that it checks a bidirectional implication in the sense that the resulting set contains *all* and *only* the required elements. There is also an analogous theorem for links, lnkCopyExecutes.

```
1 Theorem objCopyExecutes : forall o fObj fss,
2   SetContainsObject fObj o
3   /\ SetContainsObject (replacementObjects fss) o
4   /\ ~SetContainsObject (placementObjects fss) o
5   <-> SetContainsObject (objCopy fObj fss) o.
```

Listing 7: Theorem stating the correctness of objCopy using bidirectional implication

To prove the confluence of the execution function we separately prove the confluence of objCopy and lnkCopy. Listing 8 shows the confluence theorems

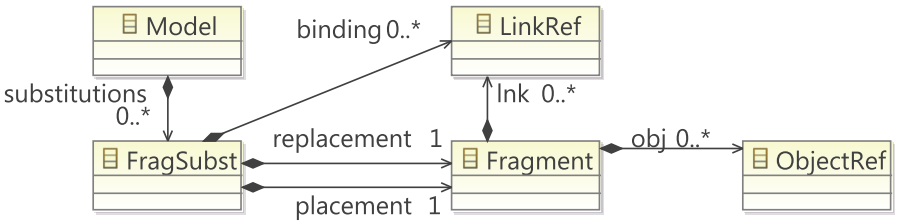

Figure 11: The metamodel of Micro CVL specified in the EMF Ecore language.

for objCopy. Theorem objCopyOnEqualFss states that the copying function is confluent with respect to the set of fragment substitutions. Theorem objCopyOfEqualSetsstates that the function is confluent with respect to the sets of object/links from which they copy the elements of the variant. Together, the two theorems along with two similar others for lnkCopy verify that executing two fragment substitution sets produces isomorphic variant models, thus execution is confluent.

```
1 Theorem objCopyOnEqualFss : forall obj  fss1  fss2 ,
2   FragSubstSetEqual fss1  fss2
3   -> ObjectSetEqual (objCopy obj fss1) (objCopy obj  fss2 ).
4
5 Theorem objCopyOfEqualSets : forall obj1  obj2  fss ,
6   ObjectSetEqual obj1  obj2
7   -> ObjectSetEqual (objCopy obj1 fss) (objCopy obj2  fss ).
```

Listing 8: Confluence theorems for objCopy

5.4. Micro CVL—a variant derivation tool

To demonstrate the verification of variant models through translation validation we implemented Micro CVL—a *small* variant derivation tool designed as a subset of CVL. In this demonstration, Micro CVL will stand for any actual variant derivation tool.

EMF facilitates the creation of Domain Specific Languages (DSL) by providing a set of tools and a meta-language, Ecore. We implemented Micro CVL to handle variability over any Ecore-based model. The metamodel of Micro CVL is presented in Fig. 11. The language is a set of fragment substitutions (FragSubst). Each fragment substitution contains one placement and one replacement fragment and a set of binding link references. The ObjectRef and LinkRef are references pointing to a base model that is an instance of an arbitrary Ecore-based domain-specific language. We can also write Micro CVL models in textual form (see Lst. 9).

A few notes on implementing fragment substitutions of Ecore models:

- EMF does not provide a way to uniquely identify objects. This is a problem because we need to know if the objects of the variant model are the same objects from the subject model. We require that all objects of the subject

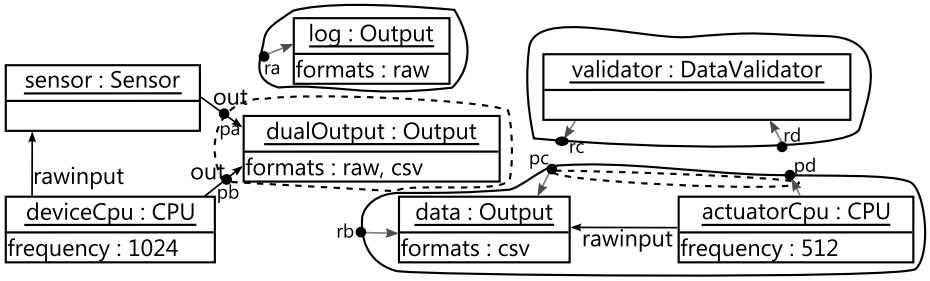

Figure 12: CVL model.

model inherit from an abstract class with an integer *id* field that is unique throughout the model. All classes in the Device metamodel inherit from the Component abstract class.

- Similarly, links cannot be uniquely identified in Ecore models. In order to reference links we identify them with a combination of the name of the metamodel relation that the link implements and the *id*s of the source and target objects. We also require that there are no multiple links between the same source and target, implementing the same relation.

5.5. Lifting and validating Ecore models

Lifting Ecore models to Coq abstract models is considerably simpler than the actual variant derivation process. After parsing the XML files and obtaining the abstract syntax trees we traverse the trees in pre-order and encode them in Coq models. It is important to notice that the Ecore models and their Coq abstractions are isomorphic, thus it is easily verifiable that the lifting is correct.

In Figure 12 we recall the Device model example from Fig. 2. We assume that the Device model has an Ecore metamodel which we do not show. We also recall the fragment substitution fs_2 of the CVL model:

$$fs_2\{\ placement\{\mathsf{pa}, \mathsf{pb}\}\ replacement\{\mathsf{rb}\}\ binding\{(\mathsf{pb}, \mathsf{rb})\}\ \}$$

We represent the fragment substitution fs_2 in Micro CVL as shown in Lst. 9.

```
1 FragSubst
2   placement Fragment
3     obj [ ObjectRef(dualOutput) ]
4     lnk [ LinkRef(sensor.out -> dualOutput) ]
5   replacement Fragment
6     obj [ ObjectRef(data), ObjectRef(actuatorCpu) ]
7     lnk [ LinkRef(actuatorCpu.rawinput -> data) ]
8   binding [ LinkRef(deviceCpu.out -> data) ]
```

Listing 9: A Micro CVL representation of fs_2

The lifting process is implemented as follows:

1. We traverse and lift the Micro CVL model to Coq.
    - whenever we encounter an ObjectRef, we search the base model for the integer *id* of the referred object;
    - whenever we encounter a LinkRef composed of the relation name and references to the source and target objects, we assign to it a new unique integer *id* and store it in a HashMap; we use this newly assigned *id* and the source and target *id*s from the subject model to lift the link.
2. We traverse and lift the output product variant Ecore model to Coq.
    - whenever we encounter an object we use the integer *id* as the abstraction;
    - whenever we encounter a link, composed of the relation name and source and target objects, we look up its assigned *id* in the HashMap created in the previous step and the source and target object *id*s as the abstraction.

Considering that the base model in Fig. 1 can be abstracted to the model in Fig. 7a, that the Micro CVL model can be abstracted to the fragment substitutions in Fig. 7b and that the derived variant model can be abstracted to the model in Fig. 7c, the lifting of the Ecore models will produce the Coq representation presented in Listing 10. Lines 1 to 1 show the complete original base model. Lines 5 to 11 show the fragment substitutions, where line 7 represents the default artifacts being copied and the other three represent the changes. Line 9 in particular represents the FragSubst from Listing 9. Lines 13 to 15 show the variant model obtained by lifting the output of the back box derivation tool.

```
1  Definition base : Model :=
2    [1; 2; 3; 4; 5; 6; 7]**[1¤3-->1; 2¤1-->2; 3¤3-->2; 4¤1-->4;
3      5¤6-->5; 6¤3-->5; 7¤7-->5; 8¤6-->7].
4
5  Definition fss : FragSubstSet :=
6    [
7       (([]**[])  ,.  ([1; 2; 3]**[1¤3-->1; 2¤1-->2; 3¤3-->2]) ., []);
8       (([2]**[2 ¤1-->2]) ,. ([4]**[])  .,  [4¤1-->4]);
9       (([2]**[2 ¤1-->2]) ,. ([5; 6]**[5¤6-->5]) ., [6¤3-->5]);
10      (([]**[])  ,.  ([7]**[])  .,  [7¤7-->5; 8¤6-->7])
11   ].
12
13 Definition variant : Model :=
14   [1; 3; 4; 5; 6; 7]**[1¤3-->1; 4¤1-->4; 5¤6-->5; 6¤3-->5; 7¤7-->5;
15     8¤6-->7].
16
17 Eval compute in bFragSubstSetIsConsistentWithModel fss base.
18
19 Eval compute in executeFss fss.
```

```
20 Eval compute in bModelEqual (EXE.executeFss fss) variant .
21
22 Eval compute in bModelConsistentWithModel variant base.
```

Listing 10: Coq representation of models

Finally, line 17 verifies that the set of fragment substitutions is consistent with the original model. Line 19 computes the simulation result by executing the Coq fragment substitution set. Line 20 evaluates whether the simulation result and the variant model are equivalent and displays the result as a Boolean value. An an extra check, line 22 checks that the variant model is consistent with the base model meaning that all links have maintained their original source and target.

6. Discussion and Related Work

Featherweight VML is closely related to CVL as it is able to express CVL models with great accuracy. Most CVL variability specifications can be reduced to features with cardinalities and the variation points are all specific cases of the fragment substitution. Featherweight VML can be seen as a generalization of OVM. We can use abstract features to group variation points together, giving OVM a tree structure while retaining the same meaning. Delta modules are almost identical to fragment substitutions. The only difference is that a delta module is guarded by an application condition over a set of features while Featherweight VML fragment substitutions are each mapped to a single feature. In order to express a Delta model without adding extra concepts we would have to change Featherweight VML's mapping function to a more general expression.

Aside from OVM, CVL and Delta Modeling, there are numerous other approaches to formalizing the elements of a variability model. Many of the variability abstractions used for software product lines, such as feature models [3] or decision models [4], are a subset of configuration modeling and knowledge-based configuration ontologies and approaches [26, 27, 28, 29]. We chose to use feature models in defining Featherweight VML simply because they are the preferred option in the industry. The mapping from features to artifacts has been specified by using feature modules [30] to wrap the artifacts pertinent to each feature and applying module composition to derive new variants; alternatively [31, 32], by annotating the artifacts with presence conditions thus specifying when each artifact must be present in a product variant. Other approaches [33] involve model transformations where each feature can both remove and add artifacts to existing models. While some formalisms are richer than feature models (e.g. Koala [34] employs a topology of components that is not a tree and interfaces between components) , a lot of these formalisms provide a fixed representation of base models (e.g. component models) and do not relate to base models specified in customized domain-specific languages or to concrete implementation artifacts such as source code [32, 35]. All these approaches bring different advantages and challenges to the domain of variability modeling, usually

compromising between expressive power and simplicity and which influences the possibility of validating actual derivation tools.

So far, most work on variability was dedicated to analyzing feature models [36, 37]. Recent work has provided valuable insight such as formalizing feature models represented in a textual language [38] or even providing full proofs in the PVS proof assistant [39]. However, the formalization is limited to feature models and do not touch on the subject of variant derivation. Czarnecki et al. [40] show how to represent the three layers of variability modeling within the single Clafer syntax. However no actual mapping to implementation artifacts is considered, just a Boolean abstraction of dependency. Such a formalization cannot directly be used as a specification of correctness for a variant derivation tool. Other works consider analyzing variability models as a whole, including checking for consistency (for instance [41, 42, 43, 44, 45]). All these methods assume correctness of the variant derivation implementation. In this work we make the first step to allow fulfilling this assumption by setting the foundation of analyzing the implementation of variability realization tools.

A crucial feature of our semantics is that it is confluent. We achieve this by identifying sufficient conditions for confluence, and adopting copying style for definition of semantics, to minimize dependencies between executions of individual variation points. Oldevik et al. [46] take a dual route and attempt to detect lack of confluence. As such they belong well to the group of works that are more interested in ensuring that models are correct than that the model manipulation tools are correct.

Since its introduction [11] translation validation has been successfully applied to compilers [47, 48], finite state machine transformations [49] or system abstractions [50]. Also, a translation validation for the LLVM compiler by abstracting the input and output to value-graphs [51] and a Coq verified translation validation by symbolic execution [52] have been proposed.

To the best of our knowledge, this is the first application of translation validation in the domain of software product lines. It is also the first implementation that is completely independent from the modeling language in which the input and output are specified. The only requirement is that the variability modeling language can be lifted to Featherweight VML. However, the lifting is considerably simpler and easier to verify than the actual variant derivation process and no extra work is required for new models and metamodels.

Coq supports automatic generation of formally verified implementations of systems (in Haskell and OCaml) out of type and function definitions. Since the lifting is an abstraction, some information from the input and output may be lost (such as attribute values of objects) which means that no concrete execution can be created automatically from an abstract execution. We believe that this use of abstraction is crucial to the success of the method. It allows implementing and growing validators incrementally, without falling into a trap of diminishing returns.

Parts of this work have been presented before. The syntax and semantics of fragment substitutions have been introduced as an extended abstract presented at the Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn. The

specification of Featherweight VML (sections 4.1–4.3) was described in [53]. This paper is a long version of the above mentioned works. It adds the entire translation validation framework around the formal semantics of Featherweight VML, including formalizations in Coq, the mechanized proofs in Coq, and implementation of translation validation on top of a home grown variability modeling tool for the Eclipse Modeling Framework.

While Micro CVL, the variability language we developed, only has demonstrative value, we also provided a proof of concept within the VARIES[11] research project. For this, we have looked at the Base Variability Resolution (BVR) language [54]. BVR is a derivative of CVL and, for the most part, it follows the same architecture and employs the same concepts. There is also a prototype implementation of a BVR tool as an Eclipse plug-in[12]. The implementation project has *over one thousand Java classes* where the product derivation code is mixed with Eclipse plug-in code and tests. By comparison, we implemented the lifting operation with just *six Java classes* with a time cost of roughly *150 man-hours* of research and development. Even though this is a rough estimate based on our own experience, it indicates that the cost of implementing translation validation for an actual tool is very small in comparison with the cost of producing the tool itself.

Once the lifting operation is implemented for a variability language, the validation technique can be applied to all projects in which the variability language is used. The nature of the product line or even of the architectural language does not influence the validation technique, meaning that the abstraction to Featherweight VML is a one-time cost. Maintenance of the abstraction is required only in case the variability language itself changes which, in our experience, does not happen very often. In case the changes to the variability language do not influence the derivation algorithm (e.g. it expands to work with a new architectural language) there are, again, no costs to the abstraction. If the changes affect the derivation process (e.g. a new feature is added) then the maintenance implies abstracting the new constructs, which can be defined as syntactic sugar in terms of fragment substitutions. In such cases it is difficult to give clear estimates of the cost, but we believe it is safe to assume that the changes to the abstraction are directly proportional to the changes of the variability language.

The translation validation technique is especially useful when creating trustworthy tools for developing safety critical systems. However, it can be applied with no extra costs to any kind of project, as long as the same variability modeling language is used. One issue that requires further investigation is the performance of the validation. Since non safety-critical projects tend to have larger and less optimized code-bases, we expect a time increase for the verified formal execution and for the equivalence check between the simulation result and the actual output model.

[11] http://www.varies.eu/

[12] GitHub repository (requires authentication): https://github.com/SINTEF-9012/bvr

7. Conclusion

We propose a translation validation of product derivation in software product lines. This technique can be applied to any variant derivation tool, but it is especially useful when creating trustworthy tools for developing safety critical systems. We formally define Featherweight VML, a compact variability modeling language, which retains the expressiveness of CVL on which it is based, but at the same time it has much simpler syntax and semantics. To our best knowledge this is the first attempt to fully formalize an entire variability model. Featherweight VML can be used as an abstraction of CVL, OVM, Delta Modeling and any other variability modeling language that satisfies the same core requirements. Our semantics processes the model in an order-agnostic manner. It is the first confluent formalization of a CVL-like language. We implemented Featherweight VML in Coq and provided proofs of the correctness and confluence of the semantics. We also demonstrated the translation validation of a black box tool, including the lifting of the input and output to Coq representations.

Acknowledgements. This work was supported by ARTEMIS JU under grant agreement n° 295397 and by Danish Agency for Science, Technology and Innovation. We thank Ina Schaefer for help with defining the semantics of Featherweight VML.

References

[1] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.

[2] P. Clements, L. M. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. rep., CMU SEI (1990).

[4] K. Schmid, R. Rabiser, P. Grünbacher, A comparison of decision modeling approaches in product lines, in: Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proc., ACM International Conference Proc. Series, ACM, 2011, pp. 119–126. doi:10.1145/1944892.1944907.

[5] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering - Foundations, Principles, and Techniques, Springer, 2005. doi:10.1007/3-540-28901-1.

[6] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond - 14th International Conference, SPLC

2010, Jeju Island, South Korea, September 13-17, 2010. Proc., Vol. 6287 of Lecture Notes in Computer Science, Springer, 2010, pp. 77–91. doi:10.1007/978-3-642-15579-6_6.

[7] CVL Joint Submission Team, Common Variability Language (CVL). OMG Revised Submission, OMG document: ad/2012-08-05 (2012).

[8] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, A. Wasowski, A survey of variability modeling in industrial practice, in: The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013, ACM, 2013, p. 7. doi:10.1145/2430502.2430513.

[9] J. Hutchinson, M. Rouncefield, J. Whittle, Model-driven engineering practices in industry, in: 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, Proc., ACM, 2011, pp. 633–642. doi:10.1145/1985793.1985882.

[10] H. Gall, Functional safety IEC 61508 / IEC 61511 the impact to certification and the user, in: The 6th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2008, Doha, Qatar, March 31 - April 4, 2008, IEEE Computer Society, 2008, pp. 1027–1031. doi:10.1109/AICCSA.2008.4493673.

[11] A. Pnueli, M. Siegel, E. Singerman, Translation validation, in: B. Steffen (Ed.), Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, 1998, Proc., Vol. 1384 of Lecture Notes in Computer Science, Springer, 1998, pp. 151–166. doi:10.1007/BFb0054170.

[12] K. Czarnecki, S. Helsen, U. W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (1) (2005) 7–29. doi:10.1002/spip.213.

[13] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, P. Y. H. Wong, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures, Vol. 6659 of Lecture Notes in Computer Science, Springer, 2011, pp. 417–457. doi:10.1007/978-3-642-21455-4_13.

[14] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, I. Schaefer, First-class variability modeling in Matlab/Simulink, in: The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013, ACM, 2013, p. 4. doi:10.1145/2430502.2430508.

[15] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, I. Schaefer, Engineering delta modeling languages, in: 17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013, ACM, 2013, pp. 22–31. doi:10.1145/2491627.2491632.

[16] Object Management Group, Meta Object Facility (MOF) Core Specification Version 2.0, OMG document: formal/06-01-01 (2006).

[17] T. Thüm, C. Kästner, S. Erdweg, N. Siegmund, Abstract features in feature modeling, in: Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011, IEEE, 2011, pp. 191–200. doi:10.1109/SPLC.2011.53.

[18] D. Clarke, M. Helvensteijn, I. Schaefer, Abstract delta modeling, in: Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010, ACM, 2010, pp. 13–22. doi:10.1145/1868294.1868298.

[19] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool features and tough decisions: a comparison of variability modeling approaches, in: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proc., ACM, 2012, pp. 173–182. doi:10.1145/2110147.2110167.

[20] A. Narayanan, G. Karsai, Specifying the correctness properties of model transformations, in: Proceedings of the Third International Workshop on Graph and Model Transformations, GRaMoT '08, ACM, 2008, pp. 45–52. doi:10.1145/1402947.1402957.

[21] T. Mens, On the use of graph transformations for model refactoring, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Vol. 4143 of Lecture Notes in Computer Science, Springer, 2005, pp. 219–257. doi:10.1007/11877028_7.

[22] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Software and System Modeling 6 (3) (2007) 269–285. doi:10.1007/s10270-006-0044-6.

[23] G. Taentzer, AGG: A tool environment for algebraic graph transformation, in: M. Nagl, A. Schürr, M. Münch (Eds.), Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1-3, 1999, Proceedings, Vol. 1779 of Lecture Notes in Computer Science, Springer, 1999, pp. 481–488. doi:10.1007/3-540-45104-8_41.

[24] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39. doi:10.1016/j.scico.2007.08.002.

[25] A. Schürr, Specification of graph translators with triple graph grammars, in: E. W. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proc., Vol. 903 of Lecture Notes in Computer Science, Springer, 1994, pp. 151–163. doi:10.1007/3-540-59071-4_45.

[26] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, M. Zanker, Transforming UML domain descriptions into configuration knowledge bases, in: Knowledge Transformation for the Semantic Web, IOS Press, 2003, pp. 154–168.

[27] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, M. Zanker, Unifying software, product configuration: A research roadmap, in: W. Mayer, P. Albert (Eds.), Proceedings of the Workshop on Configuration at ECAI 2012, Montpellier, France, August 27, 2012, Vol. 958 of CEUR Workshop Proceedings, CEUR-WS.org, 2012, pp. 31–35.
URL http://ceur-ws.org/Vol-958/paper6.pdf

[28] D. Benavides, A. Felfernig, J. A. Galindo, F. Reinfrank, Automated analysis in feature modelling and product configuration, in: J. M. Favaro, M. Morisio (Eds.), Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings, Vol. 7925 of Lecture Notes in Computer Science, Springer, 2013, pp. 160–175. doi:10.1007/978-3-642-38977-1_11.

[29] K. Czarnecki, A. Hubaux, E. Jackson, D. Jannach, T. Männistö, Unifying product and software configuration (Dagstuhl Seminar 14172), Dagstuhl Reports 4 (4) (2014) 20–35. doi:10.4230/DagRep.4.4.20.

[30] C. Prehofer, Feature-oriented programming: A new way of object composition, Concurrency and Computation: Practice and Experience 13 (6) (2001) 465–501. doi:10.1002/cpe.583.

[31] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: R. Glück, M. R. Lowry (Eds.), Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proc., Vol. 3676 of Lecture Notes in Computer Science, Springer, 2005, pp. 422–437. doi:10.1007/11561347_28.

[32] M. Janota, G. Botterweck, Formal approach to integrating feature and architecture models, in: J. L. Fiadeiro, P. Inverardi (Eds.), Fundamental Approaches to Software Engineering, 11th International Conference, FASE

2008, Budapest, Hungary, March 29-April 6, 2008. Proc., Vol. 4961 of Lecture Notes in Computer Science, Springer, 2008, pp. 31–45. doi:10.1007/978-3-540-78743-3_3.

[33] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–646. doi:10.1147/sj.453.0621.

[34] T. Asikainen, T. Soininen, T. Männistö, A Koala-based ontology for configurable software product families, in: IJCAI 2003 Configuration workshop, 2003, pp. 45–52.
URL http://www.soberit.hut.fi/pdmg/papers/ASIK03KOA.pdf

[35] M. Acher, P. Collet, P. Lahire, S. Moisan, J. Rigault, Modeling variability from requirements to runtime, in: 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011, IEEE Computer Society, 2011, pp. 77–86. doi:10.1109/ICECCS.2011.15.

[36] D. Benavides, A. R. Cortés, P. Trinidad, S. Segura, A survey on the automated analyses of feature models, in: J. C. R. Santos, P. Botella (Eds.), XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006), Octubre 3-6, 2006, Sitges, Spain, 2006, pp. 367–376.

[37] P. Schobbens, P. Heymans, J. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456–479. doi:10.1016/j.comnet.2006.08.008.

[38] A. Classen, Q. Boucher, P. Heymans, A text-based approach to feature modelling: Syntax and semantics of TVL, Sci. Comput. Program. 76 (12) (2011) 1130–1143. doi:10.1016/j.scico.2010.10.005.

[39] M. Janota, J. Kiniry, Reasoning about feature models in higher-order logic, in: Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proc., IEEE Computer Society, 2007, pp. 13–22. doi:10.1109/SPLINE.2007.36.

[40] K. Bak, K. Czarnecki, A. Wasowski, Feature and meta-models in Clafer: Mixed, specialized, and coupled, in: B. A. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers, Vol. 6563 of Lecture Notes in Computer Science, Springer, 2010, pp. 102–122. doi:10.1007/978-3-642-19440-5_7.

[41] T. Berger, S. She, R. Lotufo, K. Czarnecki, A. Wasowski, Feature-to-Code mapping in two large product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proc., Vol.

6287 of Lecture Notes in Computer Science, Springer, 2010, pp. 498–499. doi:10.1007/978-3-642-15579-6_48.

[42] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, ACM, 2013, pp. 355–364. doi:10.1145/2491956.2491976.

[43] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in: Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proc., ACM, 2006, pp. 211–220. doi:10.1145/1173706.1173738.

[44] Ø. Haugen, CVL: common variability language or chaos, vanity and limitations?, in: The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013, ACM, 2013, p. 1. doi:10.1145/2430502.2430504.

[45] F. Heidenreich, J. Kopcsek, C. Wende, FeatureMapper: mapping features to models, in: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume, ACM, 2008, pp. 943–944. doi:10.1145/1370175.1370199.

[46] J. Oldevik, Ø. Haugen, B. Møller-Pedersen, Confluence in domain-independent product line transformations, in: M. Chechik, M. Wirsing (Eds.), Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, York, UK, March 22-29, 2009. Proc., Vol. 5503 of Lecture Notes in Computer Science, Springer, 2009, pp. 34–48. doi:10.1007/978-3-642-00593-0_3.

[47] G. C. Necula, Translation validation for an optimizing compiler, in: Proc. of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, BC, Canada, June 18-21, 2000, ACM, 2000, pp. 83–94. doi:10.1145/349299.349314.

[48] T. A. L. Sewell, M. O. Myreen, G. Klein, Translation validation for a verified OS kernel, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, ACM, 2013, pp. 471–482. doi:10.1145/2462156.2462183.

[49] T. Li, Y. Guo, W. Liu, M. Tang, Translation validation of scheduling in high level synthesis, in: Great Lakes Symposium on VLSI 2013 (part of ECRC), GLSVLSI'13, Paris, France, May 2-4, 2013, ACM, 2013, pp. 101–106. doi:10.1145/2483028.2483070.

[50] J. O. Blech, I. Schaefer, A. Poetzsch-Heffter, Translation validation of system abstractions, in: O. Sokolsky, S. Tasiran (Eds.), Runtime Verification, 7th

International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers, Vol. 4839 of Lecture Notes in Computer Science, Springer, 2007, pp. 139–150. doi:10.1007/978-3-540-77395-5_12.

[51] J. Tristan, P. Govereau, G. Morrisett, Evaluating value-graph translation validation for LLVM, in: Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, ACM, 2011, pp. 295–305. doi:10.1145/1993498.1993533.

[52] J. Tristan, X. Leroy, Formal verification of translation validators: a case study on instruction scheduling optimizations, in: Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, ACM, 2008, pp. 17–27. doi:10.1145/1328438.1328444.

[53] A. F. Iosif-Lazar, I. Schaefer, A. Wasowski, A core language for separate variability modeling, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proc., Part I, Vol. 8802 of Lecture Notes in Computer Science, Springer, 2014, pp. 257–272. doi:10.1007/978-3-662-45234-9_19.

[54] Ø. Haugen, O. Øgård, BVR - better variability results, in: D. Amyot, P. F. i Casas, G. Mussbacher (Eds.), System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proc., Vol. 8769 of Lecture Notes in Computer Science, Springer, 2014, pp. 1–15. doi:10.1007/978-3-319-11743-0_1.