A Product Line of Theories for Reasoning about Safe Evolution of Product Lines

Leopoldo Teixeira
Federal University of Pernambuco
lmt@cin.ufpe.br

Vander Alves
University of Brasilia
valves@unb.br

Paulo Borba
Federal University of Pernambuco
phmb@cin.ufpe.br

Rohit Gheyi
Federal University of Campina Grande
rohit@dsc.ufcg.edu.br

ABSTRACT

A product line refinement theory formalizes safe evolution in terms of a refinement notion, which does not rely on particular languages for the elements that constitute a product line. Based on this theory, we can derive refinement templates to support safe evolution scenarios. To do so, we need to provide formalizations for particular languages, to specify and prove the templates. Without a systematic approach, this leads to many similar templates and thus repetitive verification tasks. We investigate and explore similarities between these concrete languages, which ultimately results in a product line of theories, where different languages correspond to features, and products correspond to theory instantiations. This also leads to specifying refinement templates at a higher abstraction level, which, in the long run, reduces the specification and proof effort, and also provides the benefits of reusing such templates for additional languages plugged into the theory. We use the Prototype Verification System to encode and prove soundness of the theories and their instantiations. Moreover, we also use the refinement theory to reason about safe evolution of the proposed product line of theories.

CCS Concepts

•Software and its engineering → Software product lines; Formal methods;

1. INTRODUCTION

When performing perfective and adaptive changes to software product lines [1],[1] it is important to check whether the changes were safe, that is, they preserve the observable behavior of existing products [16, 15, 5]. A theory of product line refinement [5] formalizes safe evolution in terms of a refinement notion over product lines. The term safe is used in the general sense that behavior is preserved, not specifically referring to conventional safety properties. This notion considers artifacts such as feature models (FMs) [2] and configuration knowledge (CK) [6], and instead of using the notion of refactoring, it focuses on the underlying notion of refinement [9],[2] which also captures behavior preservation but abstracts quality improvement.

The theory is also generic with respect to the different languages that we can use to describe or implement product lines. Refinement transformation templates are one of the main applications of the theory. However, to build a refinement catalog, we need to provide concrete formalizations for product line artifacts, such as FM, CK, and assets. This way we can abstract safe evolution scenarios observed from the evolution of existing product lines that use such languages.

We formalized concrete FM and CK languages, to propose refinement templates [5, 16]. However, as we formalized other FM and CK languages [15], we observed that many of them share similarities. Different FM languages share the concept of adding a variable feature or evaluating a feature expression against a configuration. This results in similar templates, and similar properties defined on the concrete formalizations. This motivated us to investigate how to explore such similarities, which resulted on the formalization of intermediate FM and CK languages. Assumptions and axioms establish the interfaces between the intermediate and concrete languages used to describe product line artifacts. The intermediate languages abstract common properties, enabling us to define templates that can be reused across a larger number of FM, CK, and asset languages.

We then structure the different formalizations as a product line of theories. Different languages correspond to different features, and products correspond to instantiations of the refinement theories, which in turn enables template reuse. Reusing specification and proofs can reduce the effort to verify them, which can be expensive [20]. This reduction is proportional to the number of languages and templates specified in the product line. Moreover, this also provides better guidance on how tools can implement support for refinement templates. We used the Prototype Verification

[1] Hereafter, product lines.

[2] Feature-oriented programming [13] uses this term in the quite different sense of overriding or adding extra behavior to assets.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-3613-0/15/07. . . $15.00
DOI: http://dx.doi.org/10.1145/2791060.2791105

System (PVS) [18] to specify and prove soundness for all of the proposed theories, theory instantiations through language formalizations, and templates.[3] Finally, we also discuss how we could use refinement templates to reason about safe evolution of the product line of theories presented here.

We organize the remainder of the text as follows. In Section 2 we review the product line refinement theory, so we then can introduce the problem with similar templates in Section 3. In Section 4, we present how we applied product line concepts to address such problem. We show how this allows reusing specification and proofs in Section 5, also discussing the associated benefits, and reasoning about evolution of such product line with the refinement theory. We discuss related work in Section 6, and conclude in Section 7.

2. PRODUCT LINE REFINEMENT

To ease understanding of the proposed product line of theories, we briefly introduce the product line refinement theory, presented in detail elsewhere [5]. For our purposes, a product line consists of three elements, that jointly generate well-formed products: (i) a variability model, describing the different product configurations; (ii) an asset mapping (AM), providing the means for referring to assets such as code, tests, and so on; (iii) a configuration knowledge [6], mapping features to assets.

We do not rely on a particular variability modeling language, as long as it represents its semantics as the set of all valid configurations. This is the case for feature models, decision models, or Kconfig models. So, for a variability model F, we only assume a generic function, denoted by $[\![F]\!]$, that defines its semantics as a set of configurations.

To enable unambiguous references to assets, instead of considering that a product line contains a set of assets, we assume it contains a mapping from asset names to assets. We assume a wf function, to distinguish arbitrary asset sets from well-formed ones. This corresponds to the notion of a well-typed program, but it may also correspond to purely syntactical or less strict semantic constraints. For most product lines, wf should be instantiated by a function that considers the well-formedness of sets of assets written in different specification and implementation languages [21].

We also rely on means of comparing assets with respect to observable behavior preservation. In particular, we assume a relation $\sqsubseteq$ establishing refinement of an asset set. This may range from imprecise lexical relations that compare assets for textual equality, to semantic relations based on bisimulation. To support stepwise refinement, this relation must be a pre-order [5], as is the case of existing notions for object-oriented programming [4]. Finally, it must be compositional, so we can apply refinements in a modular way— refining part of a well-formed product yields a refined well-formed product. This is a common property of refinement relations, that enables independent asset development. However, it is not required for all of the refinement templates we have specified using the theory.

The configuration knowledge specifies the mapping between features and implementation assets. Again, we only assume a function that is responsible for generating a product, represented as $[\![K]\!]^A_c$, for a configuration knowledge K, with an asset mapping A and a configuration c as extra parameters. In what follows, we formalize product lines.

Definition 1. *For variability model F, asset mapping A, and configuration knowledge K, we say that (F, A, K) is a product line (PL) when all of its products are well-formed:* $\forall c \in [\![F]\!] \cdot wf([\![K]\!]^A_c)$

The product line refinement notion lifts the notion of behavior preservation from products to product lines. Each product generated by the original product line must be refined by some product of the new product line. So, users of an existing product cannot observe behavior differences when using the corresponding product from the new product line. This even allows the new product line to generate more products than the original product line. In what follows, we use p and p' to denote products (well-formed asset sets); and $[\![L]\!]$ to denote the product line semantics, the function that yields its products, given by $\{[\![K]\!]^A_c \mid c \in [\![F]\!]\}$

Definition 2. *A product line L is refined by another L', denoted by $L \sqsubseteq L'$, whenever* $\forall p \in [\![L]\!] \cdot \exists p' \in [\![L']\!] \cdot p \sqsubseteq p'$

We overload $\sqsubseteq$ for product lines to denote that behavior of existing products is preserved in the new product line. Therefore, given that this is true, we have safe product line evolution. Note that the definition focuses on the generated products, instead of the configuration from the variability model. Therefore, feature names do not matter. So, users will not notice they are using products from the new product line, although developers might have to change feature nomenclature when specifying configurations.

By focusing on the common products from both product lines, we check nothing about the new products offered by the new product line. So, refinements are safe transformations in the sense that we can change a product line without impacting existing users. For example, by adding an optional feature or improving the internal structure of a feature implementation. Our theory does not aim to cover all kinds of evolution tasks. Bug fixes are not refinements, as after such changes existing users will notice behavior changes. However, the intention is to change behavior, so developers will not be able to rely on the benefits of checking refinement anyway. Such benefits only apply when the change intends to improve product line configurability or internal structure, without changing observable behavior.

Nonetheless, studies about the Linux kernel evolution [19, 14, 10] show that most of the changes consist of adding new features and modifying existing ones, while a smaller number of those changes correspond to feature removals. So, focusing on refinement covers a considerable part of these changes. Product line refinement is also compositional with respect to changes in the product line elements. Under certain conditions, variability model, configuration knowledge, and asset mapping can be independently modified [5].

3. MOTIVATING EXAMPLE

Using the refinement notion, we can safely evolve product lines. Nevertheless, it is useful to provide guidance to developers, so they do not need to reason directly about the definitions and properties. We can do so through refinement transformation templates [16, 5, 15][4] that abstract changes, such as spliting an asset, adding a new optional feature to a product line, among others. In this section, we introduce the problem with specifying and proving similar templates

[3] The PVS specification and proof files are available at http://www.leopoldomt.com/papers/pl-theories.

[4] Or only transformations.

derived from different languages, which motivated the investigation on how to reuse theories and its associated results. More details and pragmatics on refinement templates are presented elsewhere [16, 15].

The refinement theory is language-independent, so, to reason about safe evolution on concrete product lines, we need to instantiate it with concrete languages. Formalizing propositional FM [2] and compositional CK [3] languages, and then instantiating the theory with such formalizations, we derived templates specifying transformations using such languages [16, 5]. Without a systematic approach, this easily leads to a plethora of similar templates and thus repetitive specification and proof tasks, as we discuss in what follows.

Figure 1 illustrates a refinement template specifying that adding an optional feature to a product line is possible when the extra row added to the original CK is only enabled by the selection of the new optional feature [16]. This assures that products built without the new feature correspond exactly to the original product line products.

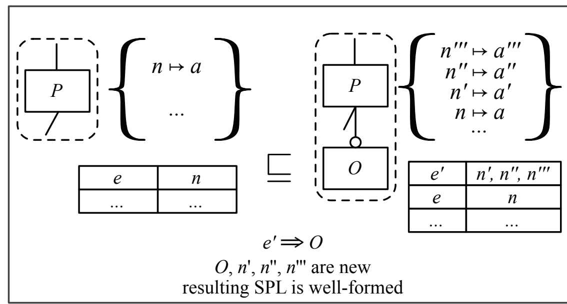

Figure 1: Add new optional feature refinement template

A template consists of a left-hand side (LHS) pattern and a right-hand side (RHS) pattern, establishing syntactic and semantic conditions for transforming a product line. Each pattern refers to meta-variables that represent product line elements. If a meta-variable appears in both sides, this means that the corresponding element remains unchanged by the transformation. In Figure 1, we observe that we require the original FM to have at least one feature, identified by P. We add a new feature O below P, together with new entries in the AM, such as $n' \rightarrow a'$. We also add a new CK item mapping the feature expression e' to the new assets. We could have variations over this template, where, for example, we add more (or less) assets.

Besides the patterns in the LHS and RHS, each transformation may have pre-conditions, detailed below the refinement symbol ($\sqsubseteq$). We can only apply a transformation when a concrete product line matches the LHS template and the pre-condition is valid for that matching. For example, Figure 1 states that the new asset names are not mapped in the existing AM, ensuring that the resulting AM is valid. Similarly, O should be a new feature that does not appear in the original FM, otherwise the resulting FM would be invalid. Feature expression e' must imply O, so the new assets are only included in products with the O feature. Finally, in this particular scenario, the template also requires well-formedness of the resulting product line, since adding an optional feature means adding the ability to generate more products, for which we cannot provide any guarantees about them through the theories [5].

Figure 2 illustrates another template, establishing that a feature expression in the CK can be replaced with an equivalent one according to the FM. This may be useful to improve the understanding of this model. We also formalize this template using the propositional FM and compositional CK languages. Notice, however, that the template does not detail the FM, indicating that we could use any FM language that has the ability to compare feature expressions. In the same way, both of the presented templates do not detail asset languages, so we can use them with any asset language that we can instantiate into the refinement theory.

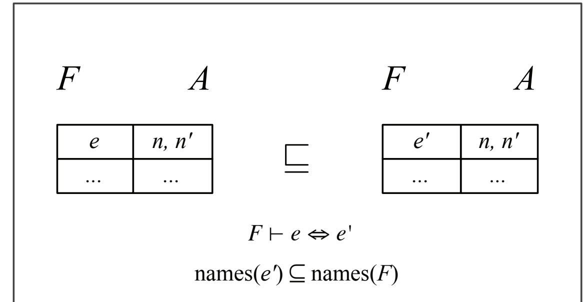

Figure 2: Replace feature expression refinement template

From Figure 2, we observe that we replace feature expression e with e'. The condition states that we can apply this transformation when e is equivalent to e', according to F, since the comparison is indexed by the FM in question. When expressions are equivalent by propositional reasoning only, we have a special case of this transformation. Moreover e' can only reference feature names that belong to F.

To prove that templates establish product line refinements, we encode them as soundness theorems by defining predicates to represent the syntactic relationships between the source and target product lines (*syntax*) and transformation pre-conditions (*conditions*). We map each template meta-variable and abstractions of product line elements to corresponding variables with associated constraints. Therefore, for each transformation we assume that the LHS product line is well-formed, and prove that the resulting product line is also well-formed and refines the original one, as follows:

$$\begin{aligned} &\forall F, A, K, F', A', K' \ldots \cdot \\ &\quad wfPL(F, A, K) \wedge syntax(\ldots) \wedge conditions(\ldots) \\ &\qquad \Rightarrow (F, A, K) \sqsubseteq (F', A', K') \wedge wfPL(F', A', K') \end{aligned} \quad (1)$$

We parameterize the *syntax* and *conditions* predicates by the meta-variables, according to the template. As discussed, for the presented templates, we do not assume a particular asset language, so it only depends on the formalized FM and CK languages. Well-formedness of the resulting product line (*wfPL*) consists of all products being well-formed according to *wf* plus additional constraints for the concrete FM and CK languages that instantiate the product line refinement theory: all feature expressions in the CK refer only to feature names in the FM, and all asset names that appear in the CK are in the domain of the AM. These extra conditions avoid repetition in the transformation pre-conditions.

These two templates could also apply for other FM and

CK languages. For example, we could change the FM language in Figure 2, as long as we are able to compare feature expressions. The same happens in Figure 1, where we do not need much detail over the CK language used. We only need a CK language that we can extend with additional items guarded by a feature expression. Changing the FM or CK language results in a new template, which in turn we must formalize and prove using the theorem structure mentioned before, with similar *syntax* and *conditions* predicates, duplicating the effort of specifying and proving transformations. Even the proof structure would be similar, differing only on the specifics of the FM and CK languages that would allow comparing feature expressions and adding new CK items.

One could argue that as an alternative, we should only define templates that rely solely on the general theory. That is, they do not depend on any concrete FM, CK, or asset language. Figure 3 formalizes the template for refining a single asset. We do not detail the actual way in which we refine the asset, as the transformation is independent from a specific asset language. The target AM has the original asset name n now associated to a', while the FM and CK remain the same. We can prove this template using compositionality properties establishing that AM refinement leads to product line refinement [5]. For a given safe evolution scenario, whenever possible we should try to prove it using the properties provided by the general theory. However, there are useful templates that we are unable to define using only the refinement theories, such as adding a new feature. This happens because such operations depend on properties that come from concrete FM or CK language.

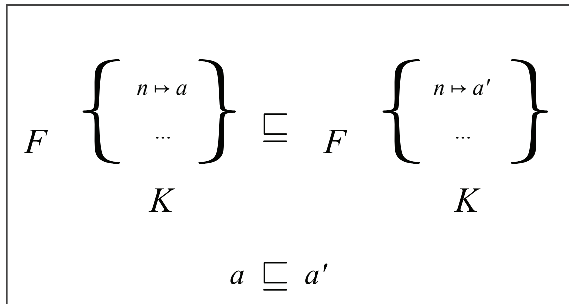

Figure 3: Refine asset refinement template

We address specific scenarios by defining templates using concrete FM, CK, and asset languages, as we show in Figure 1 and Figure 2. However, there is a considerable effort associated to specifying and proving templates in the concrete languages. Moreover, some of the templates are similar, regardless of the concrete languages, so we can explore these similarities, reducing the effort for encoding and proving templates. We do so by formalizing intermediate FM, CK, and asset languages, defining properties that abstract over the particular languages. This enables us to establish and prove templates at a higher abstraction level than those presented in this section, which then can be reused for different languages over different scenarios. Still, if that does not satisfy a particular safe evolution scenario, we must use properties and templates formalized using concrete FM, CK, and asset languages. Therefore, there is a balance that must be taken into account to define which properties should be in the intermediate levels, and which should be in the lower levels. To explore the reuse of templates and also reduce the effort of proving soundness, we structure these theories as a product line, using theory interpretation and parameterization in PVS as variability management mechanisms.

4. A PRODUCT LINE OF THEORIES

As discussed, the refinement theory is language-independent regarding artifacts such as FM (F), assets (A), and CK (K). Since it does not impose many restrictions on the structure and semantics of these elements, we might instantiate our theory in a number of ways. We want to systematically explore the similarities between languages used to describe such elements, to reduce the effort of formal specification and proof of the refinement templates. Thus, we use product line concepts to model the different formalized languages and their relationships. Figure 4 shows the FM that illustrates the different ways in which we can instantiate the product line refinement theory. Different feature selections result in different instantiations. There are cross-tree constraints, specifying, for example, that the intermediate CK language needs the intermediate FM language, as it depends on concepts such as comparing feature expressions, for example. Similarly, if we use a cardinality-based FM, we need to select cardinality-based feature expressions, since we then have to deal with cardinalities and attributes.

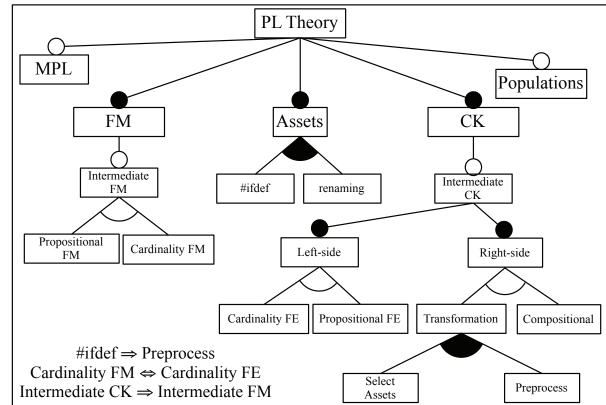

Figure 4: Feature model for the product line of theories

Figure 4 defines FM, assets, and CK as mandatory features, which reflects Definition 1, as we use these three elements to jointly generate well-formed products. We also provide intermediate and concrete formalizations for each of the artifacts because, as discussed, we can only prove some properties and templates providing more detail than what we specify in the general theory. It is important to highlight that this model only shows languages formalized in this work and is not intended to comprehend all possible languages.

We specify an intermediate language for FMs, abstracting concepts found in concrete FM languages, such as the propositional [2] and cardinality-based [7] ones. We also focus on a tabular notation for CK that associates feature expressions to assets or transformations over assets, as observed in the product lines we evaluated [21, 16, 15]. Therefore, we have subtrees for LHS and RHS, as we might have different kinds of feature expressions. We also separate, on the RHS, simple asset selection from transformations. We could include more

Feature Expression	Assets
PL Theory	SPLrefinement, set_aux_lemmas, set_comp_lemmas
Populations	Population
MPL	MultiProductLines
Intermediate FM	FMint
Intermediate FM ∨ Left-side	FeatureExpression, Configuration
Propositional FM	FeatureModel, FeatureModelSemantics, FeatureModelRefinements
Propositional FM ∨ Propositional FE	Formula_, FormulaTheory, Name
Cardinality FM	CardinalityFM
Cardinality FM ∨ Cardinality FE	Name
Assets	AssetMapping, Assets, maps
#ifdef	IfdefAssets
Renaming	RenamingAssets
Intermediate CK	CKint
Compositional	ConfigurationKnowledge, CKinst
Transformation	CKtrans, CKtInst
Select Assets	CKselect, CKselectInst
Preprocess	CKifdef, CKifdefInst
Select Assets ∨ Preprocess	CKmultipleInst
Propositional FM ∧ Compositional	SpecificFMpCKcSPL
Propositional FM ∧ Transformation	SpecificFMpCKtSPL

Figure 5: Configuration Knowledge for the product line of theories

features when dealing with implicit CKs as they occur in annotative product lines, and even to consider other kinds of transformations, for example, such as cardinality-based actions. We do not fully formalize asset languages, since this is not the main focus of our work and it would require substantial effort. We are interested on investigating the evolution of the product line as a whole, and in many cases, we can abstract the actual asset language, such as in the presented refinement templates. In the same way as we do for CK, we could define intermediate languages to abstract properties of languages, such as markup and object-oriented languages.

For the product line we propose, assets are PVS specification (*.pvs*) and corresponding proof files (sharing the name but using the *.prf* extension). Granularity of the product line [11] is coarse-grained (compositional). Therefore, the AM for this product line consists of mapping names to specification and proof files, and we use a compositional CK, which associates feature expressions to asset names.

We show the CK in Figure 5, with a single asset name in the RHS, representing the specification and its associated proof file. Most features map to a single specification and proof file, like **Intermediate CK**, but others need more files, such as **Propositional FM**. Some features share files, such as **Propositional FM** and **Propositional FE**, that use the same formalization for propositional formulae. Some files relate to the presence of two features, such as **SpecificFMpCKcSPL**, associated to the **Propositional FM** and **Compositional** features. This file contains the specification and proof of refinement templates that depend on these languages. Evaluating the CK in Figure 5 against a product configuration from the FM in Figure 4 results in a set of PVS specification and proof files.

The FM structure in Figure 4 reflects how we structure the theories. The refinement theory assumes minimal properties for FM, CK, and asset languages. We specify intermediate languages for these product line artifacts, to explore similarities and to reuse templates. For example, in Figure 5, **Intermediate CK** maps to **CKint**. The **CKint** theory defines general properties, in the form of axioms, for CK languages that map feature expressions to an action, such as asset selection or preprocessing. This enables specifying and proving templates that we can reuse with any CK language that satisfies such properties. As an example of such property, we establish that whenever we replace a feature expression by an equivalent one, CK evaluation remains the same, even without having a concrete representation of feature expressions or CK evaluation.

We use the theory interpretation mechanism of PVS [17] to show that a concrete CK language satisfies properties defined in the intermediate language. This mechanism enables us to show that a theory is correctly interpreted by another theory under a user-specified interpretation for the uninterpreted types and functions. This way, we can show that an implementation is a correct refinement of a specification. Figure 6 shows that concrete languages provide interpretations for the types and functions defined on intermediate languages. If any, axioms defined in the imported theory become proof obligations to ensure consistency of the specifications. That is, any property defined in the more abstract and high-level theories is still valid if we prove all of the proof obligations generated from the axioms. Therefore, the return on investment grows as we define more concrete languages that conform to the intermediate languages.

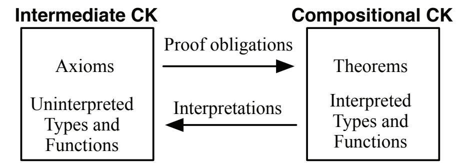

Figure 6: Theory Interpretation mechanism.

5. SPECIFICATION AND PROOF REUSE

In this section, we show how we reuse specification and proofs across theories. We present intermediate FM and CK languages (Section 5.1), that abstract properties from concrete languages. This allows us to further abstract the previously discussed refinement templates (Section 5.2). Finally, we also discuss the benefits of specifying templates at a higher level of abstraction, and discuss using the product line refinement notion to reason about safe evolution of the product line of theories that we present (Section 5.3).

5.1 Intermediate Languages

We have specified an intermediate FM theory (`FMint`), including more detail than what is required in the refinement theory, but still not consisting of a fully featured FM language. The defined properties enable us to prove templates such as Replace feature expression (see Figure 2) at a higher level of abstraction, since in this case, we only need to be able to compare feature expressions for equivalence against an FM configuration.

In the intermediate FM language, we introduce the concepts of feature expressions and the means for comparing them against product configurations (feature selections). We also specify the interfaces for other operations that are needed when formalizing the concrete FM languages, such as well-typedness of a feature expression against an FM, yielding

the set of features from a given FM, generating a feature expression from a feature, among others. Moreover, we define predicates for comparing two FMs when we add features—mandatory, optional, OR, and alternative.

Since we only establish interfaces for the functions and predicates in this theory, and do not precisely specify what they do, we define axioms to establish restriction on how concrete FM languages should specify these functions. This also acts as an interface to the function, providing guidance on how it should be specified and implemented. For example, the predicate *addOptional*, has the signature $FM \rightarrow FM \rightarrow Feature \rightarrow Feature \rightarrow boolean$. It receives the original and target FMs, together with the added feature and its parent. So, we establish that after adding an optional feature to an FM, the resulting FM generates a superset of the product configurations generated by the original FM. Additionally, the parent feature must be an existing feature in the original FM and, conversely, the added feature should not exist. This axiom prevents instantiating this function with a concrete function that would remove a feature from the original FM, for example. For the remaining functions that deal with adding features, we define similar axioms. Since the product line refinement theory only requires that FMs must define a semantics function, it is straightforward to instantiate this intermediate theory into the general theory for product line refinement. Thus, this FM language is consistent with the product line refinement theory. So, any language that instantiates it can also rely on any property that we prove according to the refinement theory.

We also formalize an intermediate CK language. We provide further detail, but, again, this is not a concrete language that we can use in practice to generate products. Rather, in this theory we abstract a set of properties observed from concrete CK languages that model the tabular notation illustrated in Figure 5. We extracted such properties by analyzing the different templates derived from our studies on product line evolution [16, 15, 5]. For proving soundness of each template, we defined auxiliary lemmas that illustrated important properties, such as replacing a feature expression in the CK by an equivalent one does not change CK evaluation. In the intermediate CK language, these lemmas become axioms. We use these axioms to prove templates at a higher level of abstraction, exploring similarities between concrete FM and CK languages. This allows reusing the templates (and proofs) for concrete FM and CK languages that comply with the intermediate languages.

Any CK language depends on FMs, as we evaluate the CK against configurations that come from the FM. Therefore, for this intermediate language, instead of defining and assuming FM types and functions, which would result in duplicated specifications, we use theory parameterization, which provides support for universal polymorphism. The intermediate CK language receives as parameter types and functions that correspond to FMs, its semantics function, and other useful auxiliary functions useful for establishing properties needed for proving templates, such as *addOptional*. We then reuse concepts and functions defined in the intermediate FM language to specify properties that we use to prove refinement templates. Any types passed as parameters must comply with the intermediate FM theory, which we do by means of theory interpretation. In what follows, we show how we specify, in PVS, this combination of theory parameterization and interpretation mechanisms.

```
CKint[FM:TYPE, [||]:FM → 𝒫[Conf], ...]:THEORY
BEGIN
IMPORTING FMint{{ ..., FM:=FM, ⟦_⟧:[||], ...}}
...
END CKint
```

We see that we pass types and functions such as `FM`, representing the feature model, and `[||]`, representing the semantics function, to the intermediate CK theory (`CKint`) as parameters. These parameters are then interpreted against the `FMint` theory, which corresponds to the intermediate FM language. So, any property that we prove with `CKint` is available for any CK language that is compliant with it, as well as any FM language that complies with `FMint`.

Although this theory specifies a generic CK based on the tabular languages previously described, we do not assume a particular representation. That is, we do not explicitly define that a CK consists of a set or a list of items where the LHS is a feature expression and the RHS is an action, that is, a set of asset names, or a transformation. Instead, for generality, we assume the CK, CK items, and the RHS of CK items as uninterpreted types, and we do not specify the CK semantics function. Additionally, we assume auxiliary functions for establishing properties, such as yielding all feature expressions and items from a CK; yielding the feature expression and action of a CK item; checking whether or not an asset name appears in a CK item. Based on such auxiliary functions, we can also define concrete functions to determine whether an asset name belongs to a CK or not, passing a CK instead of a single CK item as argument.

Prior to establishing properties, we must also establish that this CK notion is consistent with the one defined in the product line refinement theory. As we do not provide a specific representation for CKs, nor a concrete semantics function, we delegate this to the concrete CK languages that instantiate this language. We define an axiom stating that CK evaluation is compositional— refining an AM that generates well-formed products for a CK yields refined and well-formed products. Therefore, any valid instantiation of this intermediate language fulfills this property and thus is consistent with the properties defined in the general refinement theory. Then, we can define additional properties, in the form of axioms, that we can use to derive and prove refinement templates. We must demand these properties because we do not provide a fully formalized language.

5.2 Templates

We can represent a number of templates without a concrete FM or CK language. Therefore, this section presents two templates specified and proven using the intermediate FM and CK languages presented in the previous section.

Add variable feature with implementation

The transformation in Figure 1 specifies that adding an optional feature is considered a safe evolution when the added assets are only enabled by the new feature. Here, to explore the reuse and similarity between templates, reducing also the effort on proving them, we further abstract the specifics of adding a feature together with its implementation and define a single template that abstracts the three possible templates we would have for dealing with variable features, namely OR, alternative, and optional.

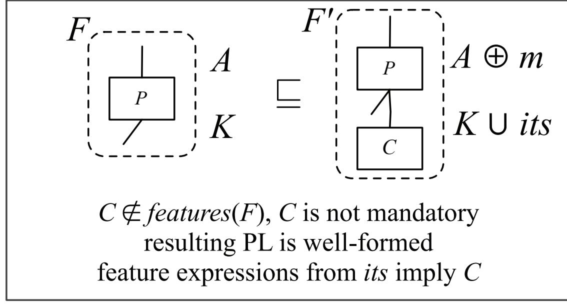

Figure 7: Add variable feature with implementation refinement template

Figure 7 illustrates the template and conditions. Although we represent the FM using the propositional language, we actually encode the template using the intermediate FM language. As with the ADD NEW OPTIONAL FEATURE refinement template from Figure 1, we only require the original FM to have at least one feature, identified in the transformation by the meta-variable P. We extend the AM arbitrarily, provided that the result is a valid AM. So, the new entries should not map names that are already mapped. Similarly, C should be a feature name that does not appear in the original FM, and its type cannot be mandatory.

As we discuss in Section 3, we encode templates by defining predicates to represent the syntactic relationships for source and target product lines (*syntax*) and by transformation pre-conditions (*conditions*), parameterized by the meta-variables, according to the template (see Equation 1). So, to prove ADD VARIABLE FEATURE WITH IMPLEMENTATION, we need to establish both *syntax* and *conditions* predicates.

The *syntax* predicate specifies syntactic similarities and differences between the source and target product lines. For this transformation we define it as follows. We add a new C feature as an optional node child of P, new items (its) to K, and new mappings (m) to A. The $\oplus$ symbol denotes that we override the asset mapping A with the names from m. We use the predicate $addVariableFeature(F, F', P, C)$ to denote the disjunction of all possible situations for this template, that is: $(addOptional(F, F', P, C) \lor addOR(F, F', P, C) \lor addAlternative(F, F', P, C))$. So, we establish a single template to handle all types of variable features that we can add to the FM, instead of establishing many templates.

For the *conditions* predicate, we require well-formedness of the target product line, since we potentially have new products, which we cannot provide any guarantee for. The new feature C cannot be an existing feature in F and new entries (m) do not use names from the asset mapping A. We also establish that the new items added to the source CK are only activated by C. Thus, for any configuration, if the feature expression of the new CK items evaluate as true, the C feature is present in that configuration.

Given the specifications for the *syntax* and *conditions* predicates, informally discussed above, we introduce the main property needed for proving soundness of the ADD VARIABLE FEATURE WITH IMPLEMENTATION transformation. Axiom 1 establishes that every product configuration c in the source product line also appears in the target product line, and that the product generated by evaluating c against the CK for both product lines is exactly the same. The intuition is that this must hold because the new items (its) are only activated by the new feature C. So, product configurations without C generate the same products, even after the change. Proving ADD VARIABLE FEATURE WITH IMPLEMENTATION means that the template applies to any concrete CK language that satisfies Axiom 1.

AXIOM 1. *Adding variable feature with guarded items does not affect CK evaluation*

For feature models F, F', asset mappings A and A', configuration knowledge K and K', set of CK items its, features P and C, and set of mappings m, if $wfCK(F, A, K)$, syntax(...), and conditions(...), then

$$\forall c \in [\![F]\!] \cdot c \in [\![F']\!] \land [\![K]\!]_c^A = [\![K']\!]_c^{A'}$$

In what follows, we prove both product line refinement and well-formedness of the resulting product line, as specified by the general theorem for proving soundness of templates (Equation 1).

Refinement: For arbitrary F, A, K, F', A', K', P, C, its, assume $wfPL(F, A, K)$, as we assume that we can only apply the transformations for product lines that are well-formed, and the *syntax* and *conditions* predicates. By Definition 2, we have to prove that $\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$. For an arbitrary $c \in [\![F]\!]$, using Axiom 1 instantiated with the variables introduced above, we have that $c \in [\![F']\!]$. Let c' be c and we have to prove that $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^{A'}$. From Axiom 1, properly instantiated with c, we also have that $[\![K]\!]_c^A = [\![K']\!]_c^{A'}$. The proof follows from this and asset set refinement reflexivity [5].

Well-formedness: Well-formedness of the resulting product line is a precondition, so the proof is trivial. We do not know details about the new product configurations (those that include C), but the precondition guarantees that the extensions to the AM and CK lead to well-formed products.

REPLACE FEATURE EXPRESSION

We also further abstracted REPLACE FEATURE EXPRESSION, illustrated in Figure 2. Figure 8 shows the resulting template, specified in a higher abstraction level. Both F and A are not detailed in the template, so we do not mention them in the *syntax* predicate. We see that the source and target CKs differ only with respect to one row. Each row is then a CK item, i_1 for the source CK and i_2 for the target CK, with the same RHS (n_s). All other CK items (its) are the same. So, we express the source CK as the union of i_1 and its, and the target CK similarly.

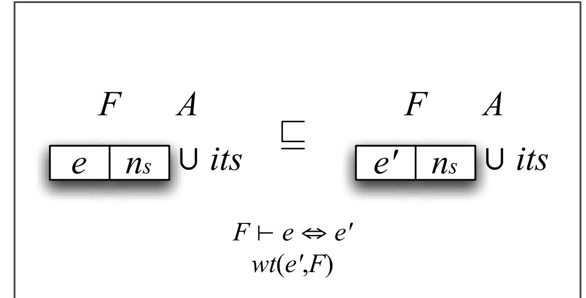

Figure 8: Replace feature expression refinement template

Notice that we do not detail the structure of the LHS or RHS of the CK. By abstracting these details, we can prove this template using the intermediate FM and CK languages, and use it with any concrete language that instantiates them. The *conditions* predicate establishes the relation between e and e', that is, the feature expressions in i_1 and i_2. We specify that e and e' are equivalent with respect to F, so all product configurations from F lead to equivalent evaluation for the feature expressions in i_1 and i_2. We also specify that the feature expression in i_2 is well-typed with respect to F. In concrete FM languages, this could mean that any feature referred to in the expression belongs to F.

Similar to Add variable feature with implementation, we also introduce an axiom to prove the Replace feature expression transformation. Such axiom also relies on the particular *syntax* and *conditions* predicate for this specific template. Axiom 2 captures the fact that replacing a feature expression by an equivalent one, according to the FM, does not affect CK evaluation. Since the expressions are equivalent, for each product configuration, their evaluation yields the same result, thus, all products remain the same. In what follows, we specify the property using the axiom, and detail both the refinement and well-formedness proofs for this template, which use the specified property.

Axiom 2. *CK evaluation is insensitive to equivalent feature expression*

For feature model F, asset mapping A, configuration knowledge K and K', CK items i_1, i_2, and the set of CK items its, if $wfCK(F, A, K)$, $syntax(...)$, and $conditions(...)$, then

$$\forall c \in [\![F]\!] \cdot [\![K]\!]_c^A = [\![K']\!]_c^A$$

Refinement: For arbitrary F, A, K, K', i_1, i_2, its, assume the *syntax* and *conditions* predicates, and $wfPL(F, A, K)$. By Definition 2, we have to prove that $\forall c \in [\![F]\!] \cdot \exists c' \in [\![F]\!] \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^A$. For an arbitrary $c \in [\![F]\!]$, let c' be c and we then have to prove that $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_c^A$. By Axiom 2 and the assumptions, properly instantiated with the variables just introduced, we have that $\forall c \in [\![F]\!] \cdot [\![K]\!]_c^A = [\![K']\!]_c^A$. The proof follows by instantiating this with the c used above and from asset set refinement reflexivity [5].

Well-formedness: Using Axiom 2, we have obtained $[\![K]\!]_c^A = [\![K']\!]_c^A$ for any $c \in [\![F]\!]$. Following the same steps, since products from the target product line refine products from the source by reflexivity, and the fact that $wfPL(F, A, K)$, the target product line is well-formed.

5.3 Discussion

The application of product line concepts to the refinement theories brings benefits when integrating more FM and CK concrete languages. For any additional concrete languages satisfying the axioms and properties defined in the intermediate FM and CK languages, the refinement templates can already be reused. That is, any refinement template proven using the intermediate languages can be automatically applied to concrete FM and CK languages, provided that they are consistent with the intermediate languages.

To prove consistency of such languages, we can follow the pattern illustrated in Figure 6, providing interpretations for the uninterpreted types in the intermediate languages. This way, axioms, such as the ones discussed in the previous section, become proof obligations, to ensure that the concrete language being formalized is actually consistent with the intermediate languages. Proving this leads to automatic reuse of any template proven that uses the axiom.

Table 1: Overview of refinement templates formalized using the intermediate languages.

Template	Gain
Add Variable Feature with Implementation	$3 * N_f * N_k$
Add Any Feature without Implementation	$4 * N_f * N_k$
Renaming	$2 * N_f * N_k$
Replace Feature Expression	$1 * N_f * N_k$
Remove dead assets	$1 * N_f * N_k$
Add dead assets	$1 * N_f * N_k$

Table 1 shows an overview of the templates formalized (so far) using the intermediate FM and CK languages. For each template, we express the benefits of abstracting it using a formula, where N_f and N_k refer to the number of concrete languages formalized that comply with the intermediate FM and CK languages, respectively. Therefore, the benefits increase as we formalize and integrate more concrete languages into the product line of theories. So, in the long run, we reduce the effort for specifying and proving templates. The number multiplying N_f and N_k refers to the number of previous templates that were abstracted by the new template. For example, as discussed, Add variable feature with implementation actually abstracts over the addition of optional, alternative, and OR features. In this case, the template formalization already provides an initial gain, that is further increased as we formalize additional languages.

Add any feature without implementation refers to an intermediate step that we observed on the evolution of some product lines [16, 15]. In some cases, it was useful to introduce new features to the FM, regardless of their type, for organization, as abstract features [24]. After introducing such feature, we can actually use it to extend the product line. For example, a report application might have only one output method, HTML, referred to in the FM as Output. We can add a new subfeature HTML, and later create new features that will form an *OR* relation with HTML. This template establishes that it is possible to add any feature to the FM, without changing the assets or the CK, since it does not affect CK evaluation. In this case, as we can add any type of feature, N_f and N_k are multiplied by 4. Renaming further abstracts the possibility of renaming features in the FM or feature expressions in the CK, so we multiply by 2. Since feature names do not matter, we establish this through an axiom stating that the actual products generated are the same, even though feature names have changed.

Finally, Add dead assets and its counterpart Remove dead assets detail the transformations where we add or remove assets that are not referred to in the CK. This can happen due to a feature retirement, where these assets would no longer be associated with any feature in the CK. This reflects the general idea for establishing this hierarchy of theories through a product line, detailed in Section 4, where we try to establish properties at the highest abstraction level possible to prove refinement templates. We establish the fact that adding or removing unused assets does not affect CK evaluation. Since no CK item refers to such assets, for each product configuration, CK evaluation yields the same result, thus, all products remain the same and we prove refinement and well-formedness.

The templates specified using the intermediate languages were proven according to asset set refinement reflexivity [5]. This reflects the fact that the syntax and conditions for applying these templates are related to changes in the FM and CK. Since we do not provide further detail over assets, we do not have any template such as SPLIT ASSET or MERGE ASSETS [16, 15], that were previously defined using concrete FM and CK languages. The proof for these templates also involved structural details over the concrete CK language and compositionality of the asset set refinement notion [5].

We have also conducted an initial evaluation over the evolution of the product line of theories proposed. Such evaluation reinforces the expressiveness of the refinement templates proposed, evaluating them in a different context than in previous studies [16, 15]. Figure 9 provides an overview of the evolution of this product line, highlighting the number of safe evolution scenarios between each version.

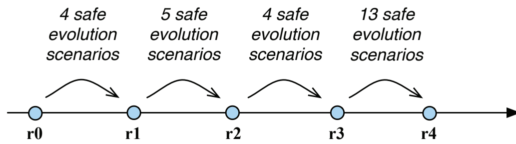

Figure 9: Evolution of the product line of theories.

The first version of the refinement theory (**r0**) was not a product line. So, we first applied an extractive adoption strategy [12] to create the product line out of the existing product. **r1** consisted of refining the general theory and directly instantiating it with concrete formalizations of FM and CK languages [5], also specifying refinement templates specific to these languages, that were evaluated for expressiveness [16, 15]. The safe evolution scenarios consisted of adding the two optional features together with their assets (ADD VARIABLE FEATURE WITH IMPLEMENTATION), and refining the existing theory (REFINE ASSET).

For the subsequent versions, we used a reactive strategy [12] to evolve this product line, extending it to contemplate more products according to our needs. For example, after **r1**, we started investigating the evolution of annotative product lines such as RGMS [15]. Thus, we needed to formalize a CK language that considered transformations, and consequently define refinement templates specific to this language. So, besides using ADD VARIABLE FEATURE WITH IMPLEMENTATION and REFINE ASSET, we also used ADD ANY FEATURE WITHOUT IMPLEMENTATION, since we added a mandatory feature only for organizing the FM.

The next evolution step consisted of formalizing the refinement theories for product populations and multi product lines [22]. In this case, we only added two optional features directly connected to the root. We again used ADD VARIABLE FEATURE WITH IMPLEMENTATION, and REFINE ASSET for changes to the product line refinement theory. We then realized that we could exploit similarities between the different FM and CK languages formalized, as discussed in Section 3. Therefore, we restructured the product line, creating new intermediate theories for FM and CK, formalizing another FM language, and restructuring the templates accordingly. By successive application of the refinement templates, in both extractive and reactive steps of the evolution of this product line, we assured safe evolution.

6. RELATED WORK

Thüm et al. conducted a survey that proposed a classification of product line analysis strategies [23]. They propose three main classification strategies, which can also be combined: product-based, feature-based, and family-based. They indicate how the analysis strategy deals with variability. Here, we do not describe product line analyses, but when specifying our product line of theories, we have properties and templates specified and proved using the three strategies. Properties proven at the refinement theory level are family-based. Intermediate and concrete languages introduce feature-based properties, in the sense that they can be reused for many products, although they are not available for the entire family. Moreover, some properties are only specified for a particular combination of concrete FM and CK languages, thus, characterizing a product-based strategy. The refinement templates that are the result of these theories describe product line transformations in different dimensions, combining different FM, asset, and CK languages.

Delaware et al. proposed product lines of theorems and proofs built from feature modules [8], realized in the Coq proof assistant. They applied product line techniques to decompose a programming language specification into theorems about properties of this language. Each module contains proof fragments, modularizing mechanized meta-theory proofs. Feature selection is manually performed, as we do here for instantiating and interpreting the theories. Different from what we do, based on a user's selection, the corresponding correctness proof is generated. In this work, we did not explore proof automation, and our theories, theorems and proofs have a different focus, regarding safe product line evolution. Moreover, our aim is similar to theirs, in the sense that we focus on a systematic approach to proving properties, to avoid rechecking proofs when instantiating the theories, although we use different mechanisms. Moreover, their formalization actually could be used as a feature in our product line, under the **Assets** feature in Figure 4.

Thüm et al. propose proof composition as a novel technique for generating correctness proofs for verifying a Java-based product line [26]. They rely on annotations for generating proof obligations, and specify feature-based proofs, to avoid specifying proofs for each program variant. The proof files are composed together in a single proof file, which is then checked. The structured approach for specifying our theories already generates proof obligations, through PVS theory interpretation mechanism. Nonetheless, we do not need to recheck proofs performed at higher abstraction levels. However, we do need to discharge the proof obligations to ensure that the interpretations are consistent with the intermediate languages and the general refinement theory.

Thüm et al. also proposed a family-based deductive verification approach [25]. They apply variability encoding to specifications, aiming to prove correctness of product lines specified using the *Java Modeling Language* (JML). Different from their work, we are not interested on proving correctness of a specific product line, but rather properties and transformations that can apply to any product line specified using the concrete languages formalized through our product line. Moreover, we use a combination of family-based, feature-based, and product-based strategies.

Staples et al. aim to evaluate productivity for proof engineering, establishing it as a key property for estimating effort on formal verification projects [20]. Their initial re-

sults indicate that proof effort is highly correlated with proof size. Our product line of theories also aims to reduce the effort on specifying and proving properties and templates, through a combination of mechanisms such as theory interpretation and parametrization. While we have not empirically assessed our productivity, we intend to do so, to provide further evidence for the benefits of applying product line concepts to structure our theories and templates.

7. CONCLUSIONS

In this work, we apply product line concepts to theories for reasoning about safe evolution of product lines, avoiding duplicated specification and proof tasks, thus, in the long run, reducing the effort for such tasks. We combine family, feature and product-based strategies [23] to achieve this goal, through mechanisms such as theory interpretation and parameterization, that could be applied in other contexts. Moreover, by using intermediate languages we can reuse templates for any concrete language that satisfies the properties established on the intermediate languages. We detail two properties and their associated refinement templates, and also discuss other templates, and the obtained gain. Finally, we use the refinement theory to reason over the safe evolution of the presented product line.

As future work, we intend to formalize more concrete languages for the product line artifacts, increasing the benefits on specifying the intermediate languages. We also intend to further investigate how to measure the return on investment of using product line concepts for the theories. Additionally, we intend to provide tool support for creating and applying refinement templates at different levels of abstraction.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering,[5] funded by CNPq (573964/2008-4) and FACEPE (APQ 0388-1.03/14). We acknowledge financial support from FACEPE grant APQ-0570-1.03/14, and CNPq scholarships and grants 477943/2013-6, 306610/2013-2.

8. REFERENCES

[1] S. Apel, D. Batory, C. Kaestner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[2] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[3] R. Bonifácio and P. Borba. Modeling scenario variability as crosscutting mechanisms. In *AOSD*, 2009.

[4] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, 2004.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2 – 30, 2012.

[6] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, 2000.

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

[8] B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *OOPSLA*, 2011.

[9] E. Dijkstra. *Notes on structured programming*. Academic Press, 1971.

[10] N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting Feature Model Changes from the Linux Kernel Using FMDiff. In *VaMoS*, 2013.

[11] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[12] C. W. Krueger. Easing the transition to software mass customization. *Lecture Notes in Computer Science*, 2290:178–184, 2002.

[13] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*. ACM, 2006.

[14] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux kernel variability model. In *SPLC*, 2010.

[15] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulezsa. Safe evolution templates for software product lines. *J. Syst. Softw.*, 106(0):42 – 58, 2015.

[16] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. In *GPCE*, 2011.

[17] S. Owre and N. Shankar. Theory interpretations in PVS. Nasa/cr-2001-211024, 2001.

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, 2001. Version 2.4.

[19] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, and P. Borba. Coevolution of variability models and related artifacts: a case study from the Linux kernel. In *SPLC*, 2013.

[20] M. Staples, R. Jeffery, J. Andronick, T. Murray, G. Klein, and R. Kolanski. Productivity for proof engineering. In *ESEM*, 2014.

[21] L. Teixeira, P. Borba, and R. Gheyi. Safe composition of configuration knowledge-based software product lines. *J. Syst. Softw.*, 86(4):1038–1053, 2013.

[22] L. Teixeira, P. Borba, and R. Gheyi. Safe evolution of product populations and multi product lines. In *SPLC*, 2015.

[23] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.

[24] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *SPLC*, 2011.

[25] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *GPCE*, 2012.

[26] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *VAST*, 2011.

[5] http://www.ines.org.br