# Porting the Software Product Line Refinement Theory to the Coq Proof Assistant

Thayonara Alves[1], Leopoldo Teixeira[1(✉)], Vander Alves[2], and Thiago Castro[3]

[1] Federal University of Pernambuco, Recife, Brazil
{tpa,lmt}@cin.ufpe.br
[2] University of Brasilia, Brasilia, Brazil
valves@unb.br
[3] Systems Development Center - Brazilian Army, Brasilia, Brazil
castro.thiago@eb.mil.br

**Abstract.** Software product lines are an engineering approach to systematically build similar software products from a common asset base. When evolving such systems, it is important to have assurance that we are not introducing errors or changing the behavior of existing products. The product line refinement theory establishes the necessary conditions for such assurance. This theory has been specified and proved using the PVS proof assistant. However, the Coq proof assistant is increasingly popular among researchers and practitioners, and, given that some programming languages are already formalized into such tool, the refinement theory might benefit from the potential integration. Therefore, in this work we present a case study on porting the PVS specification of the refinement theory to Coq. We compare the proof assistants based on the noted differences between the specifications and proofs of this theory, providing some reflections on the tactics and strategies used to compose the proofs. According to our study, PVS provided more succinct definitions than Coq, in several cases, as well as a greater number of successful automatic commands that resulted in shorter proofs. Despite that, Coq also brought facilities in definitions such as enumerated and recursive types, and features that support developers in their proofs.

**Keywords:** Software product lines · Theorem provers · Coq · PVS

## 1 Introduction

*Software product lines* (SPLs) are sets of related software systems that are systematically generated from reusable assets [1]. SPLs combine the benefits of mass customization and mass production. That is, building individual solutions from a set of reusable parts, while also tackling scale, aiming to reduce development costs and enhancing the quality of the developed products [1,8]. In this context,


G. Carvalho and V. Stolz (Eds.): SBMF 2020, LNCS 12475, pp. 192–209, 2020.
https://doi.org/10.1007/978-3-030-63882-5_12

it is important to take into account that evolving a SPL can be error-prone [10], since a single change might impact a number of products.

Previous works have defined a number of product line refinement theories [3,12,15], which provide a sound and rigorous basis to support SPL evolution, when we need to preserve the behavior of (some of the) existing products after the change. In particular, two concepts are formalized through these theories. The notion of safe evolution [3,10] denotes evolution scenarios where behavior preservation is required for all existing products in the SPL. Partially safe evolution [12], on the other hand, only requires behavior preservation for a subset of the products. In particular, in this work, we focus on the concept of *safe evolution*. These theories have been specified and proven using the *Prototype Verification System* (PVS) [11] proof assistant.

Another widely used proof assistant is *Coq*, with a large user community, which is also reflected by its solid presence in popular websites, such as GitHub repositories; more than 4000 projects returned from our search on the GitHub GraphQL API,[1] and StackOverflow questions.[2] Coq is based on the Calculus of Inductive Constructions (CIC) [14], a higher-order constructive logic. The dependent type system implemented in Coq is able to associate types with values, providing greater control over the data used in these programs.

In this work, we conduct a qualitative study whereby we ported the SPL refinement theory from PVS to Coq. Our goal is twofold: 1) to port the theory to a system used by a wider user community; but more importantly, 2) to provide a case study on this process. This enables us to reflect and investigate the similarities and differences between the two proof assistants in terms of their specification and proofs capabilities, which might be useful for the research community to better understand the strengths and weaknesses of each tool.

To make this comparison, we present some snippets of specifications from both systems, discussing similarities and differences. Moreover, we manually categorized the proof commands, which allow us to compare the proof methods at a higher granularity level. From this study, we can say that we were able to successfully port the theory to Coq, with some advantages from the point of view of usability and definition, but as the refinement theory heavily relies on sets, the PVS version ended up with a more succinct form of specification. PVS proofs also had a greater usage of automated commands than Coq, simplifying their proofs. However, this might be due to previous experience with this particular proof assistant and less experience with Coq by the authors. We have mined data from Github repositories using Coq that suggests that the Coq proofs could have been simplified using specific tactics.

The remainder of this paper is organized as follows. In Sect. 2, we present an overview of SPLs and SPL refinement. In Sects. 3 and 4, we present our comparison between the specifications and proofs in Coq and PVS. In Sect. 5, we present a discussion on lessons learned from our study. Finally, we discuss related work, and conclude pointing out future research directions in Sects. 6 and 7, respectively.

[1] https://developer.github.com/v4/explorer/.
[2] https://stackoverflow.com/questions/tagged/coq.

## 2 Software Product Lines

SPL engineering aims to increase productivity while also reducing costs, since we are not creating products from scratch. Moreover, quality might also be enhanced, since assets are possibly tested a greater number of times [1]. In this work, we adopted an SPL representation consisting of three elements: (i) a feature model that contains features and dependencies among them, (ii) an asset mapping, that contains sets of assets and asset names, (iii) a configuration knowledge, that allows features to be related to assets. In the remainder of this chapter, we introduce these elements in more detail. Variability management is an important aspect, since assets must be developed in a configurable way, to enable generating different products. Therefore, an SPL is typically structured using three different elements that are integrated to derive products.

We manage variability through features, which are usually organized into *Feature Models* (FM), describing features and their relationships [1,6]. From a particular feature model $F$, we use $[\![F]\!]$ to denote its semantics, that is, the set of valid configurations, which are used to build SPL products. Additionally, in a SPL, features are implemented by assets, which can be code, documents, configuration files, and any other artifacts that we compose or instantiate to build the different products. Since we might have different versions of the same asset, we use $A$ to refer to the *Asset Mapping* (AM), where asset names are mapped to the real assets [3]. This results in a unique mapping, which can help on eliminating ambiguities. Finally, the *Configuration Knowledge* (CK) maps features to their implementation assets, and guides product derivation. To generate a specific SPL product, given a valid FM configuration $c$, and the asset mapping $A$, we use $[\![K]\!]_c^A$ to denote the set of assets that comprise the product generated by processing the CK according to the given configuration.

Therefore, we establish that an SPL is a triple $(F, A, K)$ of such elements, that jointly generate well-formed products. Well-formedness ($wf()$) might take different meanings depending on the particular languages used for the SPL elements. For instance, it might mean that code is successfully compiling. Since we do not rely on a particular asset language, we just assume the existence of a $wf()$ function that must return a boolean value. Its concrete implementation depends on instantiating the general theory with a particular asset language.

**Definition 1.** ⟨*Software Product Line*⟩
*For a feature model $F$, an asset mapping $A$, and a configuration knowledge $K$, the tuple $(F, A, K)$ is a product line when,* $\forall c \in [\![F]\!] \cdot wf([\![K]\!]_c^A)$.

To obtain success with SPL engineering, it is important to understand the impact of changes. Thus, previous works on SPL evolution proposed ways to help developers minimize such impacts [3–5,10,12]. In this work, we deal with the *product line refinement* notion [3,15]. This notion lifts program refinement to product lines, by establishing that an SPL is refined after a change when all of the existing products have their behavior preserved. Since $[\![K]\!]_c^A$ is a well-formed asset set (a program), we use $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$ to denote the program refinement

notion. In what follows we present the main refinement notion, but we provide further details in the following section, as we discuss our formalization.

**Definition 2.** ⟨*Software Product Line Refinement*⟩
*For product lines* $(F, A, K)$ *and* $(F', A', K')$*, the latter refines the former, denoted by* $(F, A, K) \sqsubseteq (F', A', K')$*, whenever* $\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$.

The refinement theory can be used to reason about changes to an SPL, classifying them as safe evolution, if all products correspond behaviorally to the original products. However, recurring to the formal definition might be burdensome to developers, so we can leverage the theory to support developers by establishing *refinement templates* [10,12]. These templates abstract recurrent evolution scenarios which correspond to the refinement notion, that is, they are an assured way to achieve safe evolution, and they can also help on avoiding errors when evolving the SPL.

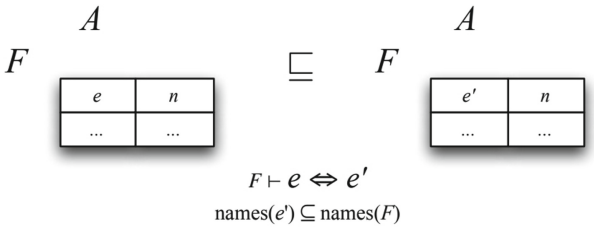


**Fig. 1.** Replace Feature Expression template.

Figure 1 illustrates the Replace Feature Expression template. We observe that a template consists of a left-hand side (LHS), corresponding to the original SPL, and a right-hand side (RHS), representing the SPL after the change. We represent the three elements of the SPL, namely the FM ($F$), AM ($A$), and CK ($K$), showing details only when needed. We also use meta-variables to denote their constituent elements. For instance, the CK consists of a feature expression $e$ mapped to an asset name $n$. If the same meta-variable appears in both LHS and RHS, this means that this element is unchanged. This template establishes that we can replace $e$ by $e'$ whenever they are equivalent according to the feature model $F$ (see condition at the bottom). Moreover, to avoid ill-formed expressions, we demand that any name belonging to $e'$ must be a valid feature name from $F$. This change can improve the readability of the CK, by replacing complex expressions with simpler ones, if they are equivalent.

## 3 Coq Formalization

In this section, we present details about our Coq formalization, in parallel with the original PVS specification,[3] when they are different. Otherwise, we only

[3] https://github.com/spgroup/theory-pl-refinement.

show the Coq code. For brevity, we are omitting some definitions. The complete formalization is available in the project repository.[4]

### 3.1 Basic Definitions

A product is described by a valid feature selection, which we call a *configuration*. Listings 3.1 and 3.2 illustrate how we represent a configuration as a set of *feature names*, as given by the `Name` type. We use *uninterpreted types*, without concrete information about it, which is an important characteristic for reasoning about arbitrary values that satisfy some specifications. The basic Coq library does not include the definition of sets. For this reason, we import the *ListSet* library for finite sets, implemented with lists, to specify sets, as is the case with *Configuration*.

**Listing 3.1:** Name and Configuration (Coq)

```
Module Name.
Require Import Coq.Lists.ListSet.
Inductive Name : Type.
Definition Configuration : Type :=
    set Name.
End Name.
```

**Listing 3.2:** Name and Configuration (PVS)

```
Name: THEORY
 BEGIN
  Name: TYPE
  Configuration: TYPE = set[Name]
 END Name
```

The validity of a configuration is given by satisfying the restrictions among features, which in our specification are expressed using propositional formulae. Such formulas are defined as a new set of data values, *enumerated types* in Coq, and *abstract datatype* in PVS. In both cases, it is necessary to provide a set of constructors that cover the abstract syntax of propositional formulae for the possible values and relations, such as true/false, feature names, negation, conjunction, and implication, which in PVS come along with associated *accessors* and *recognizers*.

**Listing 3.3:** Formula (Coq)

```
Inductive Formula    : Type :=
  | TRUE_FORMULA     : F
  | FALSE_FORMULA    : F
  | NAME_FORMULA     : Name -> F
  | NOT_FORMULA      : F -> F
  | AND_FORMULA      : F -> F -> F
  | IMPLIES_FORMULA  : F -> F -> F.

(*F = Formula*)
```

**Listing 3.4:** Formula (PVS)

```
Formula_: DATATYPE
 BEGIN
  IMPORTING Name
  TRUE_FORMULA: TRUE?: F
  FALSE_FORMULA: FALSE?: F
  NAME_FORMULA(n: Name): NAME?: F
  NOT_FORMULA(f: F): NOT?: F
  AND_FORMULA(f_0, f_1: F): AND?: F
  IMPLIES_FORMULA(f_0, f_1: F):
    IMPLIES?: F
 END Formula_
 % F = Formula_
```

Coq has a small set of built-in features, with only twelve libraries in its Prelude library versus more than one hundred theories in the PVS Prelude. When we define an inductive type, Coq automatically includes theorems, so that it is possible to reason and compute enumerated types. So, for `Formula`, Coq includes the `Formula_ind` induction principle, in addition to the `Formula_rec` and `Formula_rect` recursion principles. With these implicit definitions, we are able to define a function on the formula type and we can also use `Formula_rec` to give recursive definitions, without the `Fixpoint` command, for example.

[4] https://github.com/spgroup/theory-pl-refinement-coq.

### 3.2 Feature Model

Features are used to distinguish SPL products. A feature model is then a set of feature names, together with propositional formulae using such names. As previously mentioned, a configuration is only considered valid if it satisfies the FM formulae. The `satisfies` function is responsible for this check.

**Listing 3.5:** Valid Configuration (Coq)

```
Fixpoint satisfies (f: Formula)
    (c : Configuration) : Prop :=
 match f with
   | TRUE_FORMULA    => True
   | FALSE_FORMULA   => False
   | NAME_FORMULA n => set_In n c
   | NOT_FORMULA f1 =>
     ~(satisfies f1 c)
   | AND_FORMULA f1 f2
      => (satisfies f1 c)
         /\ (satisfies f2 c)
   | IMPLIES_FORMULA f1 f2
      => (satisfies f1 c)
       -> (satisfies f2 c)
   end.
```

**Listing 3.6:** Valid Configuration (PVS)

```
satisfies(f:Formula_,
          c:Configuration):
    RECURSIVE boolean =
    CASES f OF
      TRUE_FORMULA: TRUE,
      FALSE_FORMULA: FALSE,
      NAME_FORMULA(n): (EXISTS
       (n1:Name): c(n1) AND n1=n),
      NOT_FORMULA(f1):
       NOT satisfies(f1,c), AND_
        FORMULA(f1, f2):
         satisfies(f1,c) AND
          satisfies(f2,c),
      IMPLIES_FORMULA(f1, f2):
       satisfies(f1,c) =>
        satisfies(f2,c)
    ENDCASES
   MEASURE BY <<
```

To specify a recursive definition in PVS, we need to prove *Type-Correctness Conditions* (TCCs), to guarantee that functions always terminate. This is done by the `MEASURE` keyword that receives a well-founded order relation, to show that the recursive function is total. For recursive datatypes, PVS automatically generates such an structural order relation over `Formula`, which is provided as << in what follows the `BY` keyword. Coq employs conservative syntactic criteria to check termination of all recursive definitions, allowing recursive calls only on syntactic subterms of the original primary argument.

The semantics of the FM is given by the set of all valid configurations. In Coq, following an operational specification, we use the `genConf` function, which generates the powerset of the FM features, that is, it generates all possible configurations from a given set of features. We then use the `filter` function, which takes the FM restrictions into account, to only yield the configurations of interest, that is, the valid ones. Meanwhile, with declarative style of specification, PVS allows subtypes of the form: `A = {x: B | P (x)}`, which allows a simplified declaration of this function.

**Listing 3.7:** FM semantics (Coq)

```
Definition semantics (fm : FM)
  : set Configuration :=
   filter fm (genConf (names_ fm)).
```

**Listing 3.8:** FM semantics (PVS)

```
semantics(fm: FM): set[Configuration]=
 {c:Configuration | satImpConsts(fm,c)
  AND satExpConsts(fm,c)}
```

### 3.3 Assets and Asset Mapping

A SPL has a set of assets from which products are built. Assets are related to names through the AM. To express this, we created a theory for maps, which are basically key-value pairs, where `S` represents a key and `T` the value associated

with that key, and both are uninterpreted types. Once we have `S` and `T`, we can provide the definition of asset name, asset and map, in the `Asset` module.

**Listing 3.9:** Asset and Asset Name (Coq)

```
Definition AssetName : Type := Maps.S.
Definition Asset     : Type := Maps.T.
Definition pair_     : Type := prod S T.
Definition map_      : Type := list pair.
```

The product line refinement notion relies on an asset refinement notion. For generality, we assume this function, as the theory does not depend on a particular asset language. The basic intuition is that it returns True when refinement holds. The constructs of this function are not analogous in the two proof assistants. In Coq, we use global declarations using the `Parameter inline` command to define a function interface.

**Listing 3.10:** Asset Refinement (Coq)

```
Parameter inline assetRef :
    set Asset -> set Asset -> Prop.
```

**Listing 3.11:** Asset Refinement (PVS)

```
|- : [set[Asset],set[Asset]->bool]
```

While we do not demand a specific asset refinement notion, we require it to be a preorder. The `Orders` theory in the PVS prelude provides a nice syntactic sugar for specifying this, while we hard-coded this notion in the Coq specification.

**Listing 3.12:** Refinement is preorder (Coq)

```
Axiom assetRefinement:
 forall x y z:set Asset,
 (*reflexivity*) assetRef x x
  /\ (*transitivity*)
    assetRef x y -> assetRef y z
    ->  assetRef x z.
```

**Listing 3.13:** Refinement is preorder (PVS)

```
assetRefinement: AXIOM
 orders[set[Asset]].preorder?( |- )
```

As mentioned in Sect. 2, the AM is a unique mapping between the asset name and the asset. We also define an AM refinement notion, which is important for establishing compositionality. In this case, the original and modified AM domains must be equivalent. Any asset contained in the original AM must have a corresponding refined version in the modified AM. We observe that the Coq specification is more verbose for dealing with sets, when compared to PVS. Using Coq's dependent typing the definition could have been reduced, since we had to make explicit the `set_In an (dom am1)` premise in aMR

**Listing 3.14:** AM refinement (Coq)

```
Axiom Asset_dec :
 forall x y: Asset, {x = y} + {x <> y}.

Definition aMR (am1 am2: AM) : Prop :=
 (dom am1 = dom am2) /\
  forall (an : AssetName),
  set_In an (dom am1) ->
   exists (a1 a2: Asset),
   (isMappable am1 an a1) /\
   (isMappable am2 an a2) /\(assetRef
    (set_add Asset_dec a1 nil)
    (set_add Asset_dec a2 nil)).
```

**Listing 3.15:** AM refinement (PVS)

```
|>(am1,am2): bool =
 (dom(am1)=dom(am2) AND
  (FORALL an: dom(am1)(an) =>
    EXISTS a1,a2: (am1(an,a1))
     AND (am2(an,a2)) AND |-(a1,a2)))
```

Another detail in the Coq definition is that working with lists requires that its element types are *decidable*, due to CIC. A type has decidable equality if any two elements of that type are the same or different. Since `ListSet` uses lists to implement sets, we need to demand the following predicate $\forall xy : R, \{x = y\} + \{ x = y\}$, where $R$ is an arbitrary type for a list element. For this reason, we need to introduce axioms such as `Asset_dec` to establish that certain types are decidable. Adding axioms is a threat to validity of this study, but these axioms are mostly limited to types belonging to lists, or from things we assume to be true from the SPL refinement theory.

### 3.4 Configuration Knowledge

The CK relates features to artifacts. One way of doing this is to associate feature expressions to transformations, instead of mere asset selection. In this case, we specify the CK as a list of items, defined as the product of feature expressions and a transformation. For simplicity, we limit one transformation per item, but we could extend the theory to handle an arbitrary number of transformations.

**Listing 3.16:** CK (Coq)

```
Definition Item : Type := Formula * Transformation .
Definition CK : Type := list Item.
Parameter Inline(40) transform : Transformation -> AM-> AM -> AM.
Fixpoint semanticsCK ( ck : CK ) ( am amt : AM )
   ( c : Configuration ) : set Asset :=
match ck with
| nil = > img amt
| x :: xs = > if is_true ( satisfies ( getExp x ) c )
            then semanticsCK xs am ( transform ( getRS x ) am amt) c
             else semanticsCK xs am amt c
          end.
```

The CK semantics is defined as an interpreter that evaluates transformations to generate products. The Coq version uses pattern-matching to specify the function that runs through the list of items, applying the transformations when the expression is evaluated as true according to the configuration $c$.

### 3.5 Software Product Lines

Definition 1 states that an SPL is formed by the FM, AM and CK, jointly generating well-formed products. In PVS, we are able to use predicate subtypes to establish this fact. Using `(p)` to define a type restricts that all elements of such type satisfy the predicate `p`. This might result in proof obligations that we might need to satisfy to produce a consistent specification. In Coq, we use records. Record fields are defined with `:>`, which make that field accessor a coercion. This coercion is automatically created by Coq. Thus, in the definitions that make use of PL, the well-formedness constraint is required, as we see in Listing 3.20.

**Listing 3.17:** Software product lines (Coq)

```
Record PL: Type := {
        pls:> ArbitrarySPL;
        wfpl:> Prop; }.
```

**Listing 3.18:** Software product lines (PVS)

```
PL : TYPE = (wfPL)
```

We are then able to formalize the SPL refinement function, as in Definition 2.

**Listing 3.19:** Software product line refinement (Coq)

```
Definition plRefinement (pl1 pl2: PL): Prop :=
      (forall c1, set_In c1 (FMRef (getFM pl1)) ->
        (exists c2, set_In c2 (FMRef (getFM pl2)) /\
          (assetRef (CKSem (getCK pl1) (getAM pl1) (c1))
              (CKSem (getCK pl2) (getAM pl2) (c2))))).
```

In practice, it might not be the case that all three elements are changed in an evolution scenario [5,7]. In this sense, we prove the so-called compositionality theorems, that enable reasoning when a single one of the three elements evolves. The main idea is to establish that safely evolving one of such elements results in safe evolution of the entire SPL. We have compositionality results established for the independent evolution of each element (FM, AM, and CK), and the full compositionality theorem, that enables reasoning when all three of them evolve. It basically states that if we change the FM and CK resulting in equivalent models (this notion is provided in our repository), and the AM is refined as previously described in Listing 3.14, the resulting SPL is a refined version of the original. The formalization in Coq is analogous to that of PVS, except for the use of the *Where* command to define *pl2*, in the PVS theorem.

**Listing 3.20:** Compositionality (Coq)

```
Theorem fullCompositionality:
 forall pl fm am ck,
  equivalentFMs (getFM pl) fm /\
  equivalentCKs (getCK pl) ck /\
  aMR (getAM pl) am ->
  plRefinement pl
  {| pls:= pl2; wfpl := wfPL pl2|} /\
  wfPL ((fm ,(getAM pl)), (getCK pl)).

(*pl2 = ((fm,(getAM pl)),(getCK pl))*)
```

**Listing 3.21:** Compositionality (PVS)

```
fullCompositionality: THEOREM
 FORALL(pl,fm,am,ck): (
  equivalentFMs(F(pl),fm) AND
  equivalentCKs(K(pl),ck) AND
  |>(A(pl),am) =>
   plRefinement(pl,pl2) AND wfPL(pl2))
WHERE pl2=(# F:=fm, A:=am, K:=ck #)
```

## 3.6 Theory Instantiation and Templates

Even though we present concrete FM and CK languages in the previous section, the SPL refinement theory does not rely on a particular concrete language for FM, CK, or AM. Nonetheless, instantiating the theory with concrete languages enables us to establish refinement templates (see Sect. 2). In PVS, we do this through the theory interpretation mechanism. We use the `IMPORTING` clause to provide the parameters for the uninterpreted types and functions. PVS then generates proof obligations that we must prove to show that such instantiation is consistent.

**Listing 3.22:** Theory Interpretation (PVS)

```
IMPORTING SPLrefinement[Configuration,
WFM, Assets.Asset,Assets.
AssetName, CK, semantics, semantics]
```

In Coq, we use typeclasses. We establish the `SPL` class with interface declarations and required properties. Properties are defined as axioms or theorems. We also specify parameters for instantiating the class. As we import functions from other typeclasses, we need to handle constraints. For instance, the SPL class generates the `FeatureModel` constraints, due to the typeclass defined earlier. `FeatureModel` is satisfied by {`FM: FeatureModel F Conf`}, for example.

**Listing 3.23:** SPL Class (Coq)

```
Class SPL (A N M Conf F AM CK PL: Type) {FM: FeatureModel F Conf}
{AssetM: AssetMapping Asset AssetName AM} {ckTrans: CKTrans F A AM CK Conf}:
 Type := {
    (*====================functions======================*)
    plRefinement         : PL -> PL -> Prop;
    products             : PL -> set A
    ...
    (*===========Axioms - Lemmas - Theorems=================*)
    plStrongSubset: forall pl1 pl2: PL,
        strongerPLRefinement pl1 pl2
         -> (forall c: Conf, set_In c (FMRef (getFM pl1))
          -> set_In c (FMRef (getFM pl2)));
    ...
}.
```

We use the `Program Instance` keyword to provide concrete instance, and we must prove that each parameter satisfies the previously defined properties. Class methods must also be related to their implementation, as is the example of `plRefinement`. Finally, Coq generates obligations for the remaining fields, which we can prove in the order that they appear, using the `Next Obligation` keyword.

**Listing 3.24:** SPL Instance (Coq)

```
Program Instance Ins_SPL: SPL Asset AssetName AM Conf FM AM CK PL:= {
    plRefinement:= plRefinement_func;
    products:= products_func;
    ...
}. Next Obligation. {
    (*plStrong Subset*)
    intros.
    destruct pl1. destruct pl2.
    unfold strongerPLrefinement_func in H. specialize (H c).
    destruct c1, c0.  apply H in H0. destruct  H0. apply H0.
} Qed.
```

With such instantiation, we are then able to prove soundness of the refinement templates. That is, for each template, we prove that performing the changes as described to the original SPL, results in SPL refinement. The specification and proofs of the templates is a work in progress, so far we have two templates already fully specified and proven, and other templates already specified and with an ongoing proof. For space reasons, we refer the reader to our Github repository.

## 4 Proofs

In proof assistants, unlike automated theorem provers, we need to interact with the system to prove lemmas and theorems. For this, they provide commands—namely tactics in Coq, and rules or strategies in PVS—that act on the current proof goal, potentially transforming it into subgoals, which might be simpler to prove. Once every subgoal of the proof is dealt with, the task is finished.

Coq offers the `Search` feature. We use this command to search for previously stated lemmas/theorems that might assist proving the current goal. This prevents us from declaring other lemmas unnecessarily. As Coq's Prelude is smaller, we were unable to obtain much advantage of it, except for the lemmas from `ListSet` and other results that we had proven. PVS contains a richer Prelude, and we often used available results. For instance, nine existing lemmas were used to prove AM refinement in PVS. Nevertheless, PVS does not provide an easy search feature such as Coq, so the user needs to have prior knowledge of the theories that are in its standard library or integrate PVS with Hypatheon library[5].

Although we are porting an existing PVS specification, the proof methods often differ between the two systems. For example, Listings 4.1 and 4.2 show Coq's and PVS's proofs for the `inDom` lemma. This lemma belongs to the map theory and states that, if there is a mapping of a key $l$ to any value $r$, then the domain of that map contains $l$. In Coq, we prove the lemma by induction. The base case is solved with the `HMpb` hypothesis simplification and the contradiction tactic, since we obtain False as assumption. In the inductive case, in addition to other tactics, the `apply` tactic was used to apply the `isMappable_elim` lemma, to remove a pair from the mapping check. We also use this same tactic to transform the goal from the implications of lemmas *set_add_intro1* and *set_add_intro2*, by *ListSet*.

The `intuition` tactic was used to complete the last two subgoals. This tactic calls `auto`, which works by calling `reflexivity` and `assumption`, in addition to applying assumptions using hints from all hint databases. These calls generate subgoals that `auto` tries to solve without error, but limited to five attempts, to ensure that the proof search eventually ends. The prover has the option of increasing this number of attempts in order to increase the chances of success, as well as adding already solved proofs to the hint databases. Both should be used with care due to performance.

[5] https://github.com/nasa/pvslib.

**Listing 4.1:** inDom proof in Coq

```
Lemma inDom :
 forall am (an: AssetName) (a: Asset),
isMappable am an a -> set_In an (dom am).
Proof.
intros am0 an0 a HMpb.
induction am0.
 - simpl in HMpb. contradiction.
 - Search "isMappable".
   apply isMappable_elim in HMpb.
   inversion HMpb. clear HMpb.
   destruct H as [Heql1 Heql2].
  + rewrite Heql1. simpl. Search "set_add".
    apply set_add_intro2. reflexivity.
  + simpl. apply set_add_intro1.
    apply IHam0. apply H.
  + intuition.
  + intuition.
Qed.
```

**Listing 4.2:** inDom proof in PVS

```
inDom: LEMMA
    FORALL(m,l,r):
      m(l,r) => dom(m)(l)
(inDom 0
  (inDom-1 nil 3498485387
   ("" (skolem 1 (m l r))
    (("" (expand dom)
      (("" (flatten)
        (("" (instantiate 1 r)
        (("" (propax) nil nil))
        nil))
      nil))
    nil))
nil)
```

The PVS proof, in turn, is simpler. The basis of PVS logic is also based on set theory, which influences in this simplicity. We first perform skolemization, then we expand the definition of `dom`, which results in the proof goal `EXISTS (r: Asset): m(l,r)`. We instantiate the existential quantifier with `r`, which concludes the proof.

### 4.1 Comparing Proof Methods

To compare proof commands of both systems, we have clustered the tactics and proof rules according to their effect on the goals, as follows, with selected examples of Coq tactics and PVS rules or strategies. This is an adaptation of an existing categorization.[6]

- **Category 1 - Proving Simple Goals**: This category groups simple commands that discharge trivial proof goals.
  - **Coq:** `assumption`, `reflexivity`, `constructor`, `exact`, `contradiction`;
  - **PVS:** Simple goals are automatically solved.
- **Category 2 - Transforming goals or hypotheses**: These commands change the state of goals through simplification, unfolding definitions, using implications, among others, allowing progress in the proof process.
  - **Coq:** `simpl`, `unfold`, `rewrite`, `inversion`, `replace`;
  - **PVS:** `replace`, `replace *`, `expand`, `instantiate`, `use`, `inst`, `generalize`.
- **Category 3 - Breaking apart goals or hypotheses**: Those that split the goal or hypothesis (antecedent and consequent in PVS) into steps that are easier to prove.
  - **Coq:** `split`, `destruct`, `induction`, `case`;
  - **PVS:** `flatten`, `case`, `split`, `induct`.

[6] https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html.

- **Category 4 - Managing the local context**: Commands to add hypotheses, rename, introduce terms in the local context. There is no direct progress in the proof, but these commands bring improvements that might facilitate such progress.
  - **Coq:** `intro`, `intros`, `clear`, `clearbody`, `move`, `rename`;
  - **PVS:** `skolem!`, `copy`, `hide`, `real`, `delete`.
- **Category 5 - Powerful Automatic Commands**: Powerful automation tactics and strategies that solve certain types of goals.
  - **Coq:** `ring`, `tauto`, `field`, `auto`, `trivial`, `easy`, `intuition`, congruence;
  - **PVS:** `grind`, `ground`, `assert`, `smash`.

We could also establish other categories, but those listed here are sufficient to group all tactics and rules used in our Coq and PVS proofs.

**Table 1.** Total tactics by category

| Proof Assistant | Category 1 | Category 2 | Category 3 | Category 4 | Category 5 | Total |
|---|---|---|---|---|---|---|
| Coq | 25 | 357 | 225 | 120 | 33 | 760 |
| PVS | — | 209 | 97 | 85 | 126 | 517 |

Table 1 presents the numbers for such categories in our specifications. First, we note that the number of tactics in Coq is 47.7% higher than that of PVS, even though we often use sequences of tactics such as `destruct H0, H1`, which breaks hypotheses H0 and H1 into two others in the same step. One possible reason for these differences is the ability of PVS to automatically solve simple goals, which does not happen in Coq. Additionally, the PVS specification has taken more advantage of automatic commands. In these systems, six results are proven using the `grind` rule and, in other six proofs, this command resolved all the subgoals generated by a command of the type **Breaking apart goals and hypotheses**. The PVS documentation recommends automating proofs as much as possible. One reason may be proof brittleness, as on some occasions, updates to PVS have broken existing proofs [9].

A greater number of commands that make changes to goals and hypotheses can also be explained by a greater number of branches generated from the **Managing the local context** category. `induction`, which generates from one to six more subgoals in this formalization, was used 15 times in Coq and only five times in PVS, for example. Also, for each subgoal generated, we mark each one with bullets in Coq, increasing the number of commands in the *Breaking apart goals and hypotheses* category.

Finally, we also notice that we do not use tactics that act on other tactics - which would be a new category - such as `repeat` and `all`, which would simplify the proofs. We intend to do this in the future. We did not find this type of proof rule being used in PVS, but some commands like `skosimp` and `grind` implicitly contain control structures.

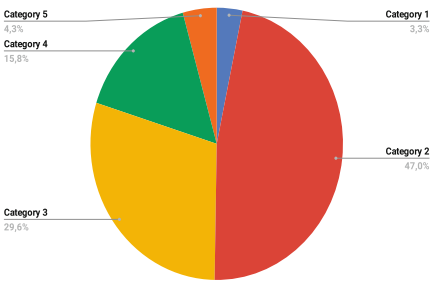


**Fig. 2.** Tactics from our Coq specification

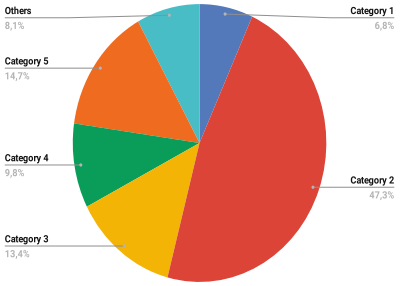


**Fig. 3.** Tactics from Github

Nonetheless, to provide some external validity for our Coq specification, we also mined Coq repositories from Github using GraphQL API v4 and PyDriller [13]. This resulted in 1.981 projects, with 65.661 `.v` files. We have also categorized the tactics from these projects, in this case also considering more categories to group a larger number of tactics. Figures 2 and 3 compare the distribution of command among the categories, for both our specification and the aggregated data from the Github projects. The percentage of **Transforming goals or hypotheses** commands is similar, but **Proving Simple Goals** appears more often among Github projects. In addition, automated commands are further explored in these projects. In projects where the `Hint` command was found, 16.43% of the tactics are automated versus 5.38% in projects that have not made such use. This suggests that we could have simplified our proofs using this command.

## 5 Discussion and Lessons Learned

The general approach of this work was to stay close to the original specification and we were able to successfully port the SPL refinement theory from PVS to Coq. However, Coq posed some difficulties during this process, as well as easing some other tasks. It is important to highlight that authors had a much stronger previous experience in PVS than Coq, so this certainly has an impact in our results. We intend to collect further feedback from experts in Coq to improve the Coq specification. Nevertheless, in general, both specifications are similar for most encodings. Most of the differences noted are presented earlier in this section and their key aspects are summarized in Table 2.

**Table 2.** Specification summary

| Section | Definition | Coq | PVS |
|---|---|---|---|
| Basic definition | Sets | ListSet library | Prelude |
| | Set of data values | Enumerated type | Abstract datatype |
| Feature model | Recursive function | Conservative syntactic criteria | TCCs; measure function |
| | FM semantics | Operational specification | Declarative specification |
| Asset and AM | AM | Maps theory | Maps theory |
| | Asset refinement | Parameter inline | Syntactic sugar for function interface |
| | Refinement is preorder | Explicit reflexivity's and transitivity's definition | Syntactic sugar from order theory |
| | AM refinement | Axioms of decidability; Explicit definition | Predicative subtype |
| SPL | PL | Record | Predicative subtype |
| | Compositionality | PL as a triple | Where command |
| Theory instantiation | Instances | Typeclasses | Theory Instantiation |

A point to be considered in favor of Coq, which contributed to the development of our specification, is its large active community and the vast amount of available information, which provides greater support to its users. There are forums that support Coq developers, as well as an active community in both StackOverflow and Theoretical Computer Science Stack Exchange.[7] We are also aware of *The Coq Consortium*,[8] which provides greater assistance to subscribing members, with direct access to Coq developers, premium bug support, among others.

Regarding the specification, Coq presented an easier way to define recursive functions. It uses a small set of syntactic and conservative criteria to check for termination, where the developer must provide an argument that is decreasing as the calls are made. In fact, there is a way to perform recursive definitions without meeting this requirement. Just specify a well-founded relation or a decreasing measure mapping to a natural number, but it is necessary to prove all obligations to show this function can terminate. On the other hand, PVS generates TCCs to ensure that the function is complete and a measure function to show this.

[7] https://cstheory.stackexchange.com/questions/tagged/coq.
[8] https://coq.inria.fr/consortium.

PVS allows partial functions, but only within total logic structures, from predicate subtypes. The definitions that made use of these subtypes made the specifications more succinct and easier to read when compared to the Coq definitions. PVS also provided syntactic sugar throughout its specification, allowing for less coding effort by the developer. We could have further leveraged Coq notations to achieve similar results. Besides that, PVS provides a greater amount of theories in its standard library. However, there is also a wide variety of Coq libraries available on the web.

The proofs are often different between the proof assistants. PVS proofs had a greater usage of automated commands like `grind`, solving some goals with just that command. It was also not necessary to worry about simpler goals. For example, the `flatten` rule not only yields a subgoal in order to simplify the goal, but it also solves simpler goals then, such as when we have `False` in the antecedent. In Coq, except in cases where automated tactics can be used, we need to explicitly use tactics such as `contradiction` to deal with False as assumption, or `reflexivity` to prove goals that are automatically discharged by PVS.

Although the proofs in Coq are longer in our specification, we noticed that important features, which give greater support to the prover, were not used. Tactics like `all` and `repeat` could have been useful to avoid repetition. The use of `Hints` and the increased search depth of the `auto` tactic may increase the chances of automated tactics being successful in their attempts.

From a usability point of view, Coq specifications and their corresponding proofs belong to the same file. In PVS, it is also necessary for the prover to be aware of control rules to go through the .prf file, such as the `undo` rule that undoes commands. In addition, it is common to lose proofs that are in these files, because of automatically renamed TCCs, for example. Finally, Coq also has search commands, either by identifying or by patterns, which might prevent the unnecessary definition of lemmas.

## 6 Related Work

Wiedijk [16] draws a comparison between 15 formalization systems, including Coq and PVS. In his work, users of each system were asked to formalize the irrationality of $\sqrt{2}$ by reducing it to the absurd. For each system, the author compare the number of lines of the specification, whether it was proven by the irrationality of $\sqrt{2}$ or by an arbitrary prime number, in addition to verifying whether the users proved the statement using their system's library. Despite the breadth of such study, the comparison among the systems is performed against a simple proof problem. This points to the need for comparison on a larger scale, in order to have the possibility to further explore the difference between these systems, specifically. This is what we have attempted to perform in this work, even though we only compare two systems. Moreover, our findings cannot be readily generalized to any mechanization using Coq or PVS, since we have only specified a particular type of theory.

Bodeveix et al. [2] formalized the B-Method using Coq and PVS. The work provides the mechanization of most constructions, showing the main aspects in the coding of the two systems for each stage of formalization. This work is similar to ours, but we also draw a comparison between the tactics and strategies that make up the proof, using data collected from Github projects to strengthen our statements.

## 7 Conclusions and Future Work

In this work, we port the existing SPL refinement theory mechanized in PVS to Coq. This theory is the basis of previous works [3,5,10,12] related to safe and partially safe evolution of SPLs, although here we only discuss the safe evolution aspect of this theory. We also compared Coq and PVS using the specifications performed on these systems, showing the differences observed in the specifications and proofs.

Through our study, we concluded that Coq's formalism and languages are sufficiently expressive to deal with and represent the different types of definitions found in the PVS mechanization. We have seen, however, that PVS provides ways to simplify most of the formalization presented, as well as proof rules that reduce the proof effort by the user. Nevertheless, we also need to emphasize that Coq brings features, such as *Hint*, tactical commands, dependent typing and advanced notations in which users of this tool can overcome this difference. Its larger community and documentation availability might provide greater support for this purpose. In addition, we must take the constant improvements made to the systems into account.

This is an initial case study on a specific project and we would benefit from conducting similar analyses on other projects. For future work, we intend to complete the proofs of the remaining SPL safe evolution templates, as well as extend our formalization to consider partially safe evolution as formalized through partial SPL refinement [12], since they are threats to the validity of this study. Besides, as mentioned earlier, the fact that the authors have more experience with PVS affects our results, being a threat to the validity of our comparison. We intend to collect feedback from Coq experts to improve our formalization. Nonetheless, our initial focus was to have a Coq specification closer to the previously proposed PVS specification. Additionally, we plan to address simplification of proofs, taking important Coq features presented in Sect. 5 into account, which was not our focus in this work.

**Acknowledgments.** This work was partially supported by CNPq (grant 409335/2016-9) and FACEPE (APQ-0570-1.03/14), as well as INES 2.0 (http://www.ines.org.br), FACEPE grants PRONEX APQ-0388-1.03/14 and APQ-0399-1.03/17, and CNPq grant 465614/2014-0. Thayonara Alves is supported by FACEPE (grant IBPG-0749-1.03/18). Vander Alves was partially supported by CNPq (grant 310757/2018-5), FAPDF (grant SEI 00193-00000926/2019-67), and the Alexander von Humboldt Foundation.

## References


1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37521-7
2. Bodeveix, J.P., Filali, M., Munõz, C.: A formalization of the B method in Coq and PVS. In: FM'99 - B Users Group Meeting - Applying B in An Industrial Context: Tools, Lessons and Techniques, pp. 32–48. Springer, Cham (1999)
3. Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. Theoret. Comput. Sci. **455**, 2–30 (2012)
4. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Autom. Softw. Eng. **23**(4), 687–733 (2015). https://doi.org/10.1007/s10515-015-0185-3
5. Gomes, K., Teixeira, L., Alves, T., Ribeiro, M., Gheyi, R.: Characterizing safe and partially safe evolution scenarios in product lines: an empirical study. In: VaMoS. Association for Computing Machinery (2019)
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (foda) feasibility study. Technical report. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990
7. Kröher, C., Gerling, L., Schmid, K.: Identifying the intensity of variability changes in software product line evolution. In: SPLC, p. 54–64. Association for Computing Machinery (2018)
8. Van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71437-8
9. Miller, S., Greve, D., Wilding, M., Srivas, M.: Formal verification of the Aamp-Fv microcode. Technical report (1999)
10. Neves, L., et al.: Safe evolution templates for software product lines. J. Syst. Softw. **106**(C), 42–58 (2015)
11. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. SRI International (2001). http://pvs.csl.sri.com/doc/pvs-language-reference.pdf, version 2.4
12. Sampaio, G., Borba, P., Teixeira, L.: Partially safe evolution of software product lines. J. Syst. Softw. **155**, 17–42 (2019)
13. Spadini, D., Aniche, M., Bacchelli, A.: PyDriller: Python framework for mining software repositories. In: ESEC/FSE, pp. 908–911. Association for Computing Machinery (2018)
14. Team, T.C.D.: The COQ proof assistant reference manual. Technical report, INRIA (2020). https://coq.inria.fr/distrib/current/refman/
15. Teixeira, L., Alves, V., Borba, P., Gheyi, R.: A product line of theories for reasoning about safe evolution of product lines. In: SPLC, pp. 161–170. Association for Computing Machinery (2015)
16. Wiedijk, F.: Comparing mathematical provers. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 188–202. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36469-2_15