UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

THAYONARA DE PONTES ALVES

PORTING THE SOFTWARE PRODUCT LINE REFINEMENT THEORY TO THE COQ PROOF ASSISTANT: A Case Study

Recife
2020

THAYONARA DE PONTES ALVES

PORTING THE SOFTWARE PRODUCT LINE REFINEMENT THEORY TO THE COQ PROOF ASSISTANT: A Case Study

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação.

Orientador: Leopoldo Motta Teixeira.

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

A474p Alves, Thayonara de Pontes
Porting the theory of software product line refinement to the Coq proof assistant: a case study / Thayonara de Pontes Alves. – 2020.
76 f.: il., fig., tab.

Orientador: Leopoldo Motta Teixeira.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de software. 2. Linguagens de programação. I. Teixeira, Leopoldo Motta (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2021 - 16

Thayonara de Pontes Alves

“Porting the Software Product Line Refinement Theory to the Coq proof assistant: A Case Study”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 27/10/2020.

BANCA EXAMINADORA

_____

Prof. Dr. Gustavo Henrique Porto de Carvalho
Centro de Informática/ UFPE

_____

Prof. Dr. Rohit Gheyi
Departamento de Sistemas e Computação / UFCG

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE
(Orientador)

To my parents.

ACKNOWLEDGMENTS

Ao meu Deus e eterno Pai, fonte verdadeira da luz e da ciência, que pacientemente dirige o curso da minha vida e me faz perseverar no que me é proposto, a revelia de mim mesma.

À minha família, especialmente aos meus pais, Marcos e Dalva, pelas constantes abnegações, orações e provisões que me fizeram chegar até aqui. Sou grata ao Júnior, meu irmão, e também à Manu, a irmã que ele me deu, pelo companheirismo ao longo desses anos e por me presentearem com o meu sobrinho Gael, que já enche a minha vida de tantos sentimentos bons. Também sou imensamente grata às minhas avós, Beatriz e Severina (in memoriam), à Larissa, ao Matheus, às minhas tias e demais primos, pelo amor, carinho e apoio jamais negados. Além desses, gostaria de agradecer à Jessica, minha amiga de longa data, que mesmo não compartilhando de laços sanguíneos, sempre foi uma verdadeira irmã para mim. Vocês todos são graça abundante revelada a mim diariamente.

Ao Alessandro, meu noivo e melhor companhia que poderia desejar em ter. Agradeço por me ajudar a focar em meus objetivos, seja pelas palavras de incentivo, pelo silêncio oportuno, pelo ombro, pelos atos de serviço e até mesmo pelas danças. A leveza que você me traz faz tudo ficar mais divertido, fácil ou suportável.

Ao professor Leopoldo, sempre preocupado com o que lia, fazia e escrevia, mas também em como estava me sentindo. Agradeço pelas recomendações, pela compreensão e o apoio prestado em todas as etapas desse programa. Também sou grata pela sua exemplaridade acadêmica, ministerial e pessoal, que tanto me inspira.

Agradeço também ao professor Vander e ao Thiago, pelas valiosas contribuições dadas a essa pesquisa.

Aos professores Rohit e Gustavo, por terem aceitado o convite para participar da banca e sugerirem melhorias para esse trabalho.

À tanta gente bacana que conheci em mais esses dois anos no Centro de Informática. Agradeço aos meus amigos, Karine e Pedro, pelas boas recordações que irei levar desse tempo. Torço e vibro pelas conquistas de vocês como se fossem minhas.

Aos meus irmãos da igreja A Ponte, onde encontro as pessoas mais imperfeitamente extraordinárias que conheço. Sou grata pelos ensinamentos e direcionamentos recebidos, pelo cuidado mútuo que há, pelo partilhar e pelo incentivo de minha vocação. Ao Gui, Kaliu, Kauê, Wando e demais integrantes do Alicerce, sou grata também pela oportunidade de servir com vocês. Vocês são fera!

Por fim, agradeço à FACEPE, pelo incentivo financeiro à pesquisa.

"It is said that an argument is what convinces reasonable men and a proof is what it takes to convince even an unreasonable man." (CRAIG, 1994)

ABSTRACT

Proofs are not a simple task to be performed. Some barriers are also put in place when it comes to checking them, as there are proofs that are so specialized that few people can even understand them or so long that few have time to check them. Computers have been an ally in this sense, as they support those who deal with it, automating all or part of the process, in addition to performing the verification of the proof steps. In this context, we have proofs assistants that are capable of generating some proof steps automatically, but that still need the collaboration of a user to conduct the process. There are a variety of proof assistants, however, with different purposes. A better understanding of strengths and weaknesses regarding these systems can lead to a choice that means less effort for formalization and proof, for instance. In this work, we codified a specification of the software product line refinement theory in the Coq proof assistant. This theory guarantees that we are not introducing errors or changing the behavior of existing products in a product line during an evolution, ensuring a safe evolution. This theory has been specified and proved using the Prototype Verification System (PVS) proof assistant. Nevertheless, 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. This specification includes specific models such as Feature Model, Asset Mapping, and Configuration Knowledge, as well as instantiation using Typeclasses and formalizing templates that can be used in SPL evolution scenarios. Moreover, due to the fact that this theory has already been formalized in the PVS, 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.

RESUMO

As provas não são uma tarefa simples de serem realizadas. Algumas barreiras também são postas quando se trata de verificá-las, uma vez que existem provas que são tão especializadas que poucas pessoas são capazes de entendê-las ou tão longas que poucas dispõe de tempo para checá-las. Os computadores vêm sendo um aliado nesse sentido, pois dão suporte para aqueles que lidam com isso, automatizando todo ou parte do processo, além de realizar a verificação dos passos de provas. Nesse contexto, temos os assistentes de provas que são capazes de gerar alguns passos de provas de forma automática, mas que ainda precisam da colaboração de um usuário para conduzir o processo. Existem uma variedade de assistentes de provas, porém, com finalidades diferentes. Um melhor entendimento de pontos fortes e fracos a respeito desses sistemas podem levar a uma escolha que signifique em um menor esforço de formalização e prova, por exemplo. Nesse trabalho, codificamos uma especificação da teoria de refinamento de linha de produtos de software no assistente de provas Coq. Essa teoria dá a garantia de que não estamos introduzindo erros ou alterando o comportamento dos produtos existentes de uma linha de produtos durante uma evolução, assegurando uma evolução segura. Esta teoria foi especificada e comprovada usando o assistente de prova *Prototype Verification System* (PVS). No entanto, um outro assistente de prova, *Coq*, tem se tornado cada vez mais popular entre pesquisadores e desenvolvedores e, dado que algumas linguagens de programação já estão formalizadas em tal ferramenta, a teoria do refinamento pode se beneficiar do potencial de integração. Dessa forma, neste trabalho, apresenta-se um estudo de caso sobre a portabilidade da especificação PVS da teoria de refinamentos para Coq. Esta especificação inclui modelos específicos, tais como *Feature Model*, *Asset Mapping* e *Configuration Knowlegde*, como também a instanciação usando *Typeclasses*, além da formalização de *templates* que podem ser usados em cenários de evolução de SPL. Adicionalmente, pelo fato dessa teoria já ter sido formalizada no PVS, este trabalho compara os assistentes de prova com base nas diferenças observadas entre as especificações e as provas dessa teoria, proporcionando algumas reflexões sobre as táticas e estratégias utilizadas para compor as provas. Como resultado, de acordo com este estudo, o PVS forneceu definições mais sucintas do que o Coq, em vários casos, bem como um maior número de comandos automáticos bem-sucedidos que resultaram em provas mais curtas. Apesar disso, Coq também trouxe facilidades nas definições, como tipos enumerados e recursivos, e recursos que dão suporte aos desenvolvedores em suas provas.

Palavras-chaves: Linhas de produtos de software. Provadores de Teorema. Coq. PVS.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AM	ASSET MAPPING
CC	CALCULUS OF CONSTRUCTIONS
CIC	CALCULUS OF INDUCTIVE CONSTRUCTIONS
CK	CONFIGURATION KNOWLEDGE
CTC	CROSS TREE CONSTRAINTS
FM	FEATURE MODEL
SPL	SOFTWARE PRODUCT LINE

CONTENTS

1 INTRODUCTION

The question "What is proof?", and variations on it, have been discussed for some time, and many answers have been proposed, because different communities may agree on different answers to this question. In particular, in Mathematics, proofs are absolutes (GEUVERS, 2009). This means that we can rely on the statements once they have been proved. Moreover, there are many methods that can be used to prove the statements. In Formal Proofs, we assume that some hypotheses are true and make use of known facts and the deduction rules of logic to reach the conclusion (JIMÉNEZ; SÁNCHEZ, 2010). Every logical inference made must be checked all the way back to the fundamental axioms of mathematics, which is why there is a need for a precise syntax to make proofs easily understood by provers and checkers.

Formal proofs are often constructed with the help of computers, allowing some or all of the proof steps to be generated automatically. These proofs can be checked automatically, also by the computer. In this context, proof assistants provide a formal language to write mathematical definitions, theorems and executable algorithms. In contrast to automated theorem proving, these proof management systems need a man to guide the process. However, the fact of having a user interacting with the system increases expressiveness, making it possible to set-up a generic mathematical theory in such a system (GEUVERS, 2009).

The Mizar, PVS, HOL, HOL4, Light, Isabelle/HOL and Coq tools are examples of this type of system. Each of them has its particularities, which may lead to less or greater effort, depending on the situation, both with the development of proofs and formalizations. However, there are not so many studies that present cases of specific formalizations with the intention of making comparisons between these systems. In our work, we chose to formalize the *Product Line Refinement Theory*, which guarantees the management of the evolution of SPL artefacts over time and ensuring the consistent integration of the changes in all affected products of the SPL (POHL; BÖCKLE; LINDEN, 2005).

We consider the choice of this theory because it requires several types of definitions, such as data value, uninterpreted types, recursive functions, axioms, lemmas, theorems, in addition to several properties to be proved. In addition, one of the authors of this work was responsible for the formalization in PVS, which provides a specification language based on higher-order logic and a proof checker based on the sequent calculus that combines automation (decision procedures), interaction, and customization (strategies).

In addition to PVS, another proof assistant is *Coq*. Coq is based on the *Calculus of Inductive Constructions* (CIC) (TEAM, 2017), a higher-order logic that is constructive and very expressive. Since the introduction of the CIC and the first implementation of Coq, very few issues have been found in the underlying theory of Coq (TABAREAU, 2020). Coq also provides a specification language, called *Gallina*, to represent the usual types and programs of programming languages, capable of formalizing both mathematical definitions, algorithms and

also theorems, and the command language is called Vernacular (BERTOT; CASTRAN, 2010). The dependent type system implemented in Coq is able to associate types with values, providing greater control over the data used in these programs.

Coq has a large user community, which is also reflected by its solid presence in popular websites, such as GitHub repositories and StackOverflow questions.[1]. We found more than 4.000 projects returned from our search on the GitHub GraphQL API.[2] There are several projects in different areas that have made use of this tool. An example is in the mathematics field. Georges Gonthier, as well as workmates from Microsoft Research and INRIA, have proved the Feit-Thompson theorem (GONTHIER, 2011). Another example is related to the companies Gemalto and Trusted Logic. These companies obtained the highest level of certification (EAL 7) for their formalization of the security properties of the JavaCard platform (CHETALI, 2008).

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; 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.

This study then tackles questions like *"Is Coq expressive enough to deal with the definitions made in the specification of the SPL Refinement Theory in PVS?"*. Furthermore, regarding the comparison, *"Are there different efforts regarding the specifications and the proofs made in the two proof assistants?"*. The answer to the first question is achieved from our formalization of the SPL refinement theory through direct mapping of the same specification in PVS. For the second case, we will use the comparison between the systems.

To make this comparison, we present some snippets of specifications from both systems, discussing similarities and differences. Moreover, we manually categorized the used proof commands, which allowed 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. We have mined data from GitHub repositories using Coq that suggests that the Coq proofs could have been simplified using specific tactics.

Therefore,this work provides the specification in Coq of concrete theories such as Feature Model, Asset Mapping and Configuration Knowledge, in addition to the general theory of SPL and the creation of instances of this theory with the concrete ones, which are used for

1 <https://stackoverflow.com/questions/tagged/coq>
2 <https://developer.github.com/v4/explorer/>

formalization of the SPL safe evolution templates. Additionally, we contributed with a comparison between Coq and PVS based on the specification of the SPL theory performed on these two systems.

The remainder of this work is organized as follows:

- **Chapter 2** presents an overview of SPLs and SPL refinement;
- **Chapter 3** presents other essential concepts used throughout this work, such as the concept of proofs, proof assistants, in addition to characteristics about PVS and Coq;
- **Chapter 4** presents the formalization of the SPL refinement theory and our comparison between the specifications and proofs in Coq and PVS. Besides, it presents a discussion on lessons learned from our study;
- **Chapter 5** presents the final considerations of this work, discusses related work, and future work.

2 SOFTWARE PRODUCT LINES

Nowadays, in order to meet the different desires and needs of customers, mass customization has brought individualism back to the focus of production, without neglecting large-scale production capacity (APEL et al., 2013). This combination of interests was made possible through the reuse of parts in which their combinations generate different products. The benefits of this approach make the idea of reuse present in different industrial segments. For example, in Figure 1, there are some steps to place an order in a food delivery App. Customers have customization options by choosing what they want to put in or take out of their burritos. This brings a sense of individuality, despite users being limited in their options.

According to Clements and Northrop (CLEMENTS; NORTHROP, 2001) a software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. The most obvious benefit from SPL development is increased productivity, but from SPLs we also have the advantages of reducing costs, since we are not creating a single product from scratch, and also a better quality of what is delivered to the customer, since assets are typically more exposed and tested by being used in different products (APEL et al., 2013).

In software development, mass customization is present with SPL adoption that enables the generation of related software products from reusable assets (POHL; BöCKLE; LINDEN, 2005; BORBA; TEIXEIRA; GHEYI, 2012). SPLs have gained considerable momentum in recent years, both in industry and in academia. One case is the Core Flight Software System (CFS) SPL. Launched as open source in 2015, it was developed by NASA's Goddard Space Flight Center (GSFC) and has been serving as a base for software that records space experiments that without using it would take years to complete (SUKHWANI et al., 2016). Companies and institutions such as Hewlett Packard, General Motors, Boeing, Nokia, Siemens, Toshiba and Philips also apply product line technology with great success. A prominent example from the open-source community that can be considered as an SPL is the Linux kernel with more than 11.000 features (TARTLER et al., 2012).

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 (BORBA; TEIXEIRA; GHEYI, 2012; PASSOS et al., 2015). In the remainder of this chapter, we introduce these elements in more detail.

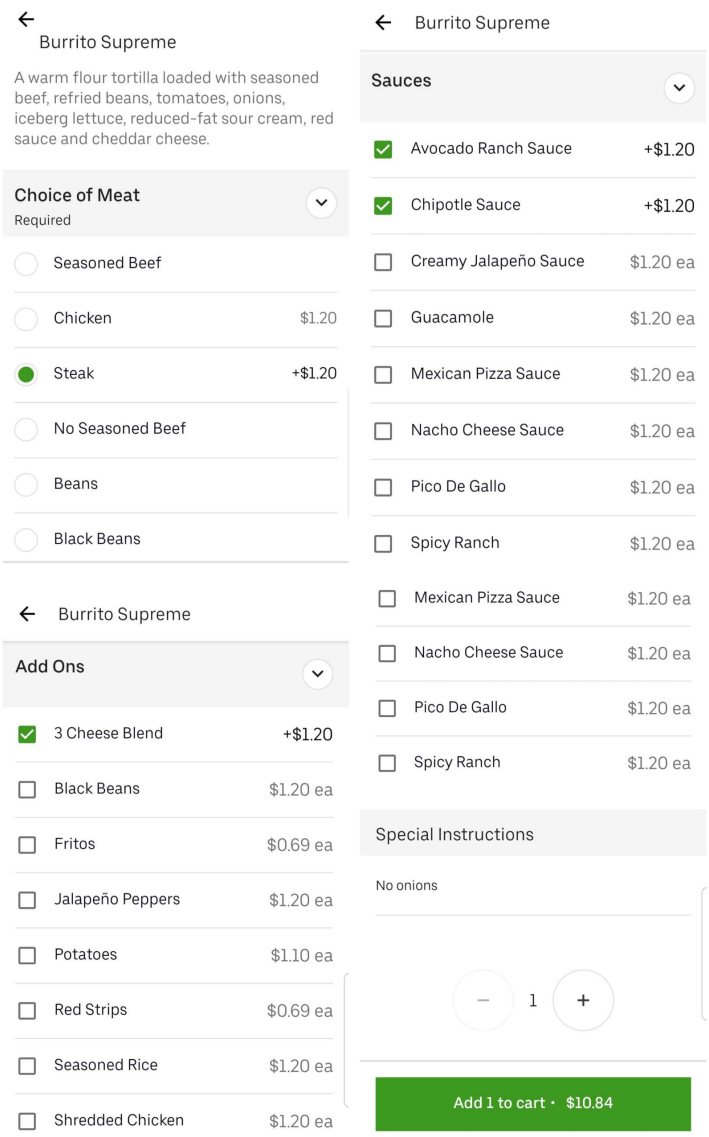

Figure 1 – STEPS TO PLACE AN ORDER IN A FOOD DELIVERY APP.

2.1 FEATURE MODEL

Variability management is an important aspect in SPLs, because software development must take the fact that they are configurable into account, meaning that the assets must be developed in order to enable the generation of different products within the SPL. This requires a way to distinguish members of a family, which is achieved through the concept of features. Although the concept of a feature is inherently hard to define precisely (APEL et al., 2013), we assume a feature to be a prominent and distinctive user visible characteristic of a system (APEL et al., 2013; KANG et al., 1990).

Feature Model (FM) were proposed as part of the FODA (Feature-Oriented Domain Analysis) method (KANG et al., 1990) and are used to capture the combinations of features in SPLs. The relationships among features are usually displayed as a tree in FMs, whose nodes are feature names and a specific notation is used to specify the relationships.

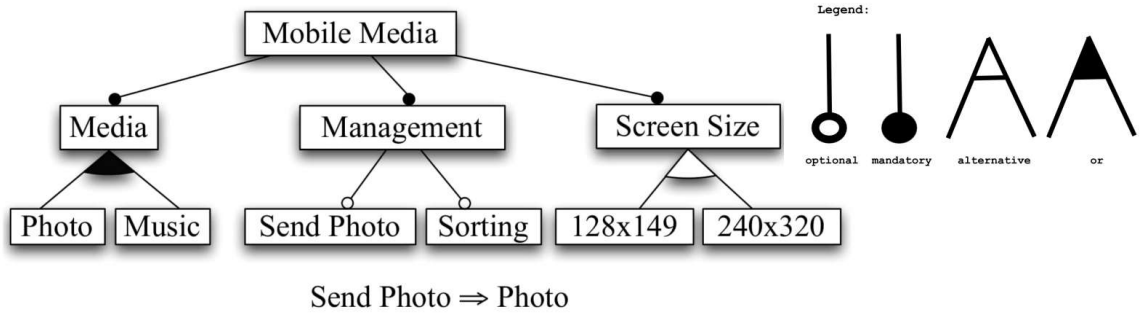

Figure 2 – FEATURE MODEL (NEVES ET AL., 2015).

Figure 2 describes the MOBILE MEDIA FM. Each software product must have the MEDIA, MANAGEMENT and SCREEN SIZE **mandatory features**. The SEND PHOTO and SORTING features may or may not be present, since they are **optional features**. In addition, only one screen size can be selected for a given product in this line, as these are **alternative features**. Each system will also feature at least one of the **or-inclusive features**: PHOTO and MUSIC.

Restrictions are imposed graphically to have valid configurations. In this work, we define a configuration as a set of feature names. Therefore, some valid configurations, according to the FM in Figure 2, are:

- {MOBILE MEDIA, MEDIA, PHOTO, MANAGEMENT, SORTING, SCREEN SIZE, 128X149}
- {MOBILE MEDIA, MEDIA, MUSIC, MANAGEMENT, SEND PHOTO, SCREEN SIZE, 128X149}
- {MOBILE MEDIA, MEDIA, MUSIC, PHOTO, MANAGEMENT, SORTING, SCREEN SIZE, 128X149}

We can increase the expressiveness of FMs by inserting representation of relationships among features of different branches called *cross tree constraints* (CTC). With the CTC we have SEND PHOTO and PHOTO related in the FM of Figure 2. Through the formula, SEND PHOTO => PHOTO, we know that PHOTO must be selected whenever SEND PHOTO is. This means that {MOBILE MEDIA, MEDIA, MUSIC, MANAGEMENT, SEND PHOTO, SCREEN SIZE, 128X149} is an invalid configuration, since the constraint expressed by the formula is not satisfied.

2.2 ASSET MAPPING

A SPL has a set of assets on which a shared family of systems is built. Assets are the concretization of the features, introduced in Section 2.1, and can be documents, XML files, images, and other artifacts that we compose or instantiate in different ways to specify or build

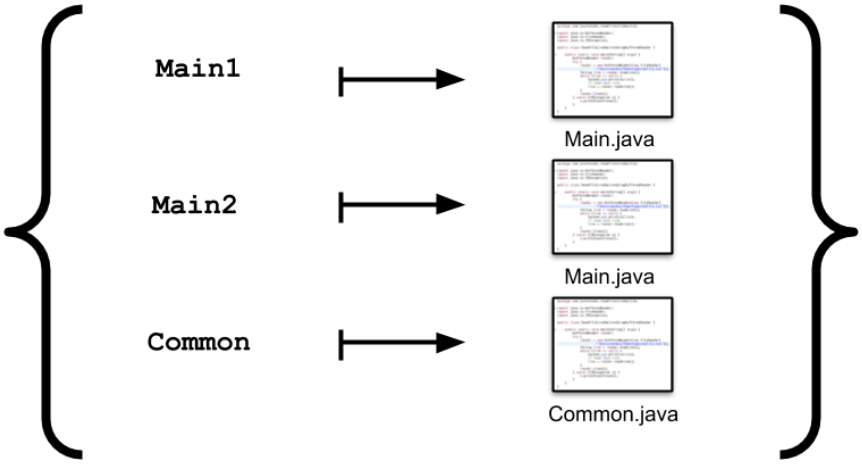

Figure 3 – ASSET MAPPING (NEVES ET AL., 2015).

the different products. But for simplicity, for the examples and concepts, we focus on code assets that can be specified in any language. In Mobile Media, for example, there must be software components that allow to take photos, play music and send photos. So, when creating a product with the following configuration {MOBILE MEDIA, MEDIA, PHOTO, MANAGEMENT, SORTING, SCREEN SIZE, 128X149} we can select the assets that will be part of this product, excluding from this selection the asset whose name is MUSIC, responsible for the execution of songs in this product. In an Asset Mapping (AM), these asset names are mapped to real assets, being a unique mapping and able of eliminating ambiguities. In the example in Figure 3, we have the names of the assets on the left side related to their respective assets, on the right side. There are two MAIN.JAVA, however, one is related to the asset name MAIN1 and the other to MAIN2, which is possible because these are different files in different locations.

2.3 CONFIGURATION KNOWLEDGE

The Configuration Knowledge (CK) defines restrictions on how the variation of system families should be composed for the derivations of products by mapping features to their implementation. To generate a specific product from an SPL, given a valid FM configuration, the processing of a CK generates the sets of source codes needed to build the corresponding product. 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.

An explicit and compositional representation of a CK is given in Figure 4. It is represented as a table, showing a mapping of feature expressions to a set of asset names. It is the asset name because it is the path to the asset itself. When selecting PHOTO, for example, PHOTO.JAVA, among others that do not appear in Figura 4 must be present in the final prod-

Mobile Media	MM.java,...
Photo	Photo.java,...
Music	Music.java,...
Photo ∨ Music	Common.aj,...
Photo ∧ Music	App.aj,...
...	...

Figure 4 – CONFIGURATION KNOWLEDGE (NEVES ET AL., 2015).

uct. If PHOTO and MUSIC are selected, COMMON.AJ, MUSIC.JAVA, PHOTO.JAVA, among others, are part of the product.

Once we have introduced the triple of such elements, we give the formal definition of SPL, where all products are well-formed. 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, a configuration knowledge K, the tuple (F, A, K) is a product line when, $\forall c \in [[F]] \cdot wf([[K]]_c^A)$.

2.4 PRODUCT LINE EVOLUTION

Despite promised benefits, adopting the SPL approach involves considerable barriers that not all companies are prepared to face (APEL et al., 2013). One of these barriers is associated to the costs involved in establishing the product line, which exceeds the values of traditional strategies (LINDEN; SCHMID; ROMMES, 2007). To compensate for this disadvantage, SPLs must evolve, whether motivated by new customer requirements, either foreseeing future needs of the company or to launch a new product. Looking at Figure 5, only from three products, which is the point where costs are equivalent, the costs of single systems grow faster than those of the system family, which makes SPL a more profitable strategy than traditional ones.

Nevertheless, some challenges arise when an SPL evolves. Each product is a configuration resulting from a number of possible configurations. This variability can escalate until

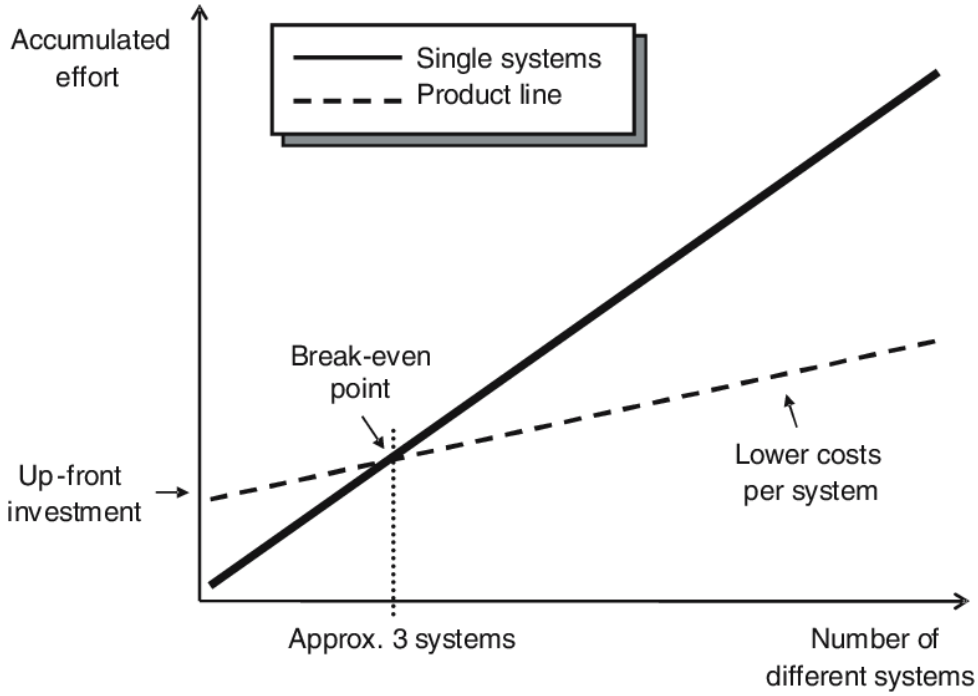

Figure 5 – SPL X SINGLE SYSTEM (POHL; BÖCKLE; LINDEN, 2005).

it becomes an almost impossible task to manage, if we consider manually performing this task. This may mean introducing bugs, modifying the behavior of products from the original line, compromising the benefits promised by this approach (NEVES et al., 2015).

The success of evolution lies in understanding the impacts of changes, and several studies have proposed ways to help developers to minimize such impacts (BORBA; TEIXEIRA; GHEYI, 2012; SAMPAIO; BORBA; TEIXEIRA, 2019; NEVES et al., 2015; GOMES et al., 2019; BüRDEK et al., 2015). Some of these works are divided between the concepts of *safe and partially safe evolution*, and consider the notion of SPL refinement. This notion was based on the idea of program refinement. SPL refinement leverages the notion of program refinement, allowing the addition of new products and preserving the behavior of existing ones. This idea is formalized through a refinement theory (BORBA; TEIXEIRA; GHEYI, 2012), that has been encoded and proved using the PVS, a proof assistant.

In general, what differentiates safe evolution from partially safe evolution is how many products in the original product line will have their behavior preserved. It is a prerequisite for safe evolution that all products correspond behaviorally to the original PL products (NEVES et al., 2015). Thus, users of an original product cannot observe behavioral differences when using the same functionalities as the corresponding product in the new SPL.

For the evolution to be safe, FM and CK must be taken into account. Figure 6 shows the refinement in adding the optional Copy feature to Mobile Media SPL. This change results in twice as many products in the new SPL, that is, for each existing configuration before the change, it is possible to include the name of the new feature. And we can say that this is a safe evolution because half of them behave exactly like the original products.

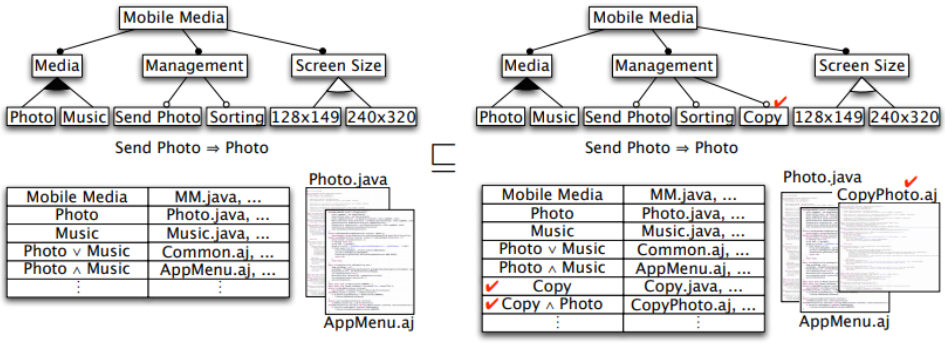

Figure 6 – ADDING AN OPTIONAL FEATURE REFINEMENT (NEVES ET AL., 2015).

Partially safe evolution, in turn, arises from the need to support developers in scenarios where at least one product will have its behavior modified, for example, a bug fix. So, in these cases, partially safe evolution only requires behavior preservation for a subset of the existing products in a SPL (SAMPAIO; BORBA; TEIXEIRA, 2019).

In particular, in this work, we focus on the concept of *safe evolution*, which is formalized through the *product line refinement* notion. This notion lifts program refinement to product lines, by establishing that an SPL is a refinement 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 chapter, 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* $\forall c \in [[F]] \cdot \exists c' \in [[F']] \cdot [[K]]_c^A \sqsubseteq [[K']]_{c'}^{A'}$.

2.5 CAPTURING EVOLUTION SCENARIOS BY TEMPLATES

Studies on safe and partially safe evolution have resulted in templates that are abstractions of recurrent practical evolution scenarios in SPL. Neves et al. (2015) initially analyzed the evolution scenarios of the TaRGet SPL, resulting in 11 evolution scenarios and a set of safe evolution templates that abstract, generalize and factorize the analyzed scenarios and that can be used by developers in charge of maintaining SPLs. Benbassat, Borba and Teixeira (2016) on the other hand, conducted a study using an industrial system developed in Java with 400 KLOC, revealing the need for new templates to address feature extraction scenarios. Sampaio, Borba and Teixeira (2019) has defined templates based on these refinement templates by changing conditions to allow partially safe changes, finding evidence that these templates could cover a number of practical evolution scenarios by analyzing commits from the Linux and Soletta systems. Gomes et al. (2019) also analyzed commits from the open

source project Soletta, but this time, the goal was to characterize the evolution of the SPL as a whole, measuring to what extent the evolution history in safe compared to partially safe.

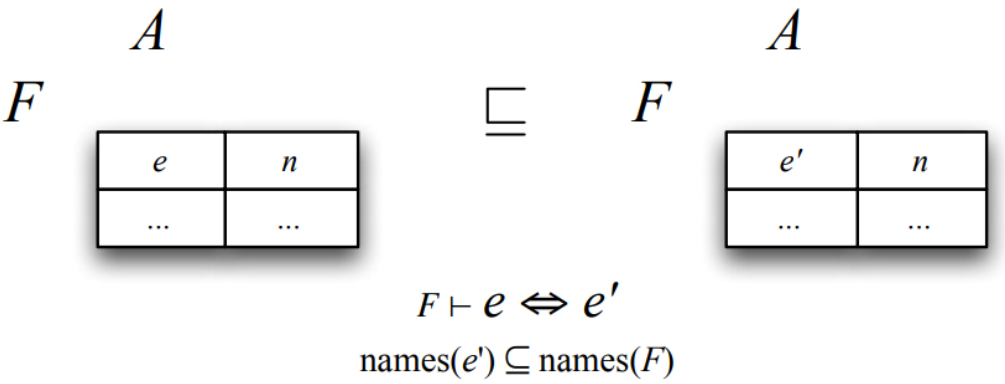

Figure 7 – TEMPLATE OF REPLACE FEATURE EXPRESSION (NEVES ET AL., 2015).

Templates are useful because they help developers not to introduce defects in the product line by evolving the SPL manually, and it can result in the development of tools to help developers make their changes. It is not required that developers have knowledge of the theory behind these templates, but only that they understand what the templates suggest for them to make the change safely.

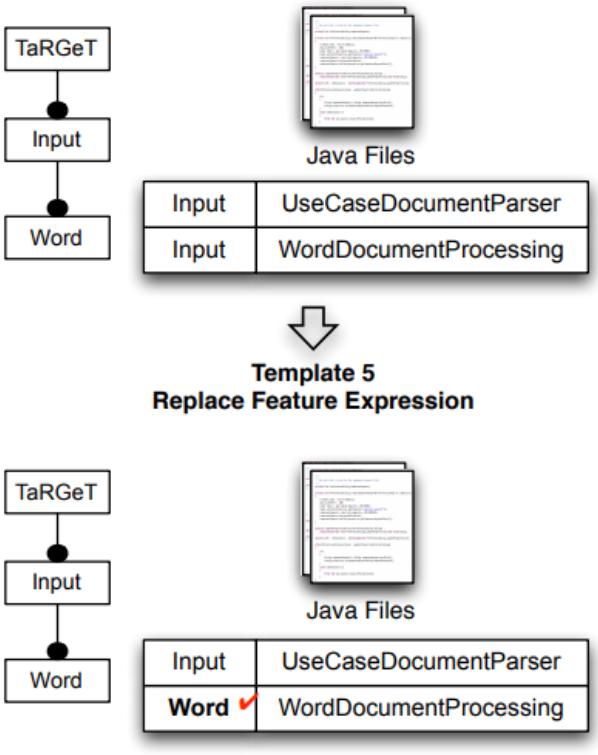

Figure 8 – TEMPLATE OF REPLACE FEATURE EXPRESSION EXAMPLE (NEVES ET AL., 2015).

To understand templates, it is necessary to know what is common for all templates. The templates show the representations of the FMs, AMs and CKs. Those in the left side correspond to abstractions that capture properties of the initial SPL (LHS) (SAMPAIO; BORBA; TEIXEIRA, 2019), and in the right side we have the evolved SPL (RHS) (SAMPAIO; BORBA; TEIXEIRA, 2019). The FM representation only shows the features involved or affected by the evolution. AM is represented by two square brackets, with meta variables to indicate the name of the assets and the assets. Finally, we have the CK composed of two columns, in the first column there are the meta variables for the expressions of features, and in the second, the meta variables for the name of the assets. If there are meta variables F, A and K, which represent FM, AM and CK, respectively, it means that these models remain unchanged after evolution.

If the developer wants to substitute a feature expression, according to the template shown in Figure 7, the formula $F \vdash$ e $\iff$ e' shows that all product configurations of a feature model F should lead to an equivalent evaluation for the feature expressions in *e* and *e'*. The developer is also required to use a new feature expression that references F names. This change is useful because it improves the readability of CK and FM.

Figure 8 illustrates how we can use this template. In this case, the developer uses the template to change the feature expression related to the WORDDOCUMENTPROCESSING asset from INPUT to WORD. This is possible because as WORD is under INPUT, selecting the first means that the second is also selected.

3 PROOF ASSISTANTS

In the previous chapter we gave an introduction to SPLs and their structure. In this chapter, we continue to provide the relevant concepts for understanding our work. In the current chapter, we give an introduction to proofs in general, to proof assistants and focus on providing some characteristics of the Coq proof assistant.

3.1 THE ROLES OF PROOF

Mathematicians, philosophers, educators, computer scientists and others interested in mathematics, try to agree on what a mathematical proof actually means. Despite this, we are all aware that a proof follows a sequence of steps that leads us to reach a goal. More specifically, according to the Webster's dictionary definition, a proof is *"the process or an instance of establishing the validity of a statement especially by derivation from other statements in accordance with principles of reasoning"* (MERRIAM-WEBSTER, 2011). These derivations start from established axioms or other theorems that have already been proven.

The several different roles that proofs play can also be an aggravating factor in the task of defining proof. Villiers identified a non-exhaustive list of proof roles (VILLIERS, 1990). According to him, a proof can assume the role of verification, explanation, systematization, discovery, intellectual challenge, and communication. Other works consider a proof as a justification of definitions. Among these, the roles of *verification* and *explanation* are the main ones. Geuvers (GEUVERS, 2009) agrees with this statement when he defines two roles for a proof:

- **A proof convinces the reader that the statement is correct**.
- **A proof explains why the statement is correct**.

The first point is to verify the correctness of basic steps and how they are combined to have the proof as a whole, validating it. Additionally, a theorem can have different explanations, each of which shows a new point of view of why the proof is correct. This last role helps us to counter the complaints that it does not make sense to prove theorems that "everyone knows" or that have been proven in the past, and allow us to justify why new proofs of old theorems are valuable (VILLIERS, 1990). Mathematicians often value a new proof of a well-established theorem for its explanatory power. In this way, we also have the challenge of explaining what can be obtained with a proof beyond the knowledge that the resulting theorem is true.

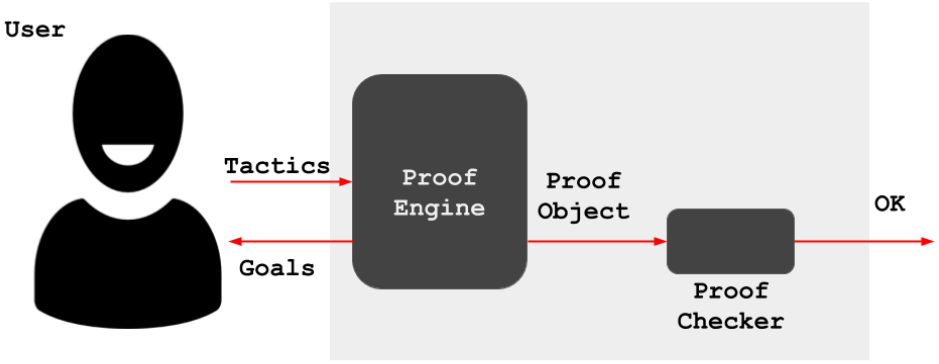

Figure 9 – PROOF DEVELOPMENT IN A TYPE THEORY BASED PROOF ASSISTANT (GEUVERS, 2009).

3.2 PROOF ASSISTANTS

Proof assistants (also known as "interactive theorem provers") are systems that automatically generate some proof steps, but depend on the man to guide the process from the beginning to the conclusion, providing support for provers and checkers in their tasks that, without them, could be tedious and with greater possibilities of introducing errors. The need for interaction with a user has the advantage of greater expressiveness in specifications, so any theory can be formalized in this type of system. These systems usually come with an interface where the user enters commands to manipulate the current goal, that is, what one intends to prove at the moment, or hypotheses. These goals are transformed into other subgoals that are easier to prove, and once the prove of each one of them is finished, the proof is concluded as a whole. This is basically a trial and error task (YANG; DENG, 2019). The user analyzes the feedback of the tool to conclude whether to proceed or to go back a step to try another command.

Proof assistants deal with formal proofs that, in every step, have their construction verified according to the fundamental axioms of mathematics. So, internally, there must be a logical structure to allow formal verification, so that we can trust the tool. In the case of proof assistants, there is the proof engine, which is responsible for building the proof-term from the inputs of user (GEUVERS, 2009). These proof-terms are not visible to the user, but they are objects that are checked during the construction of the proof.

3.3 PVS

PROTOTYPE VERIFICATION SYSTEM (PVS) is a tool that provides an expressive specification language based on higher-order logic (CROW et al., 1995), including, by default, types such as

numbers, records, matrices, sets, lists, as well as a mechanism for defining abstract types. This tool is also an interactive environment for conducting formal proofs, able to solve various objectives in an automated way. PVS provides a collection of powerful primitive inference procedures, which include propositional and quantifier rules, induction, rewriting, data and predicate abstraction, and symbolic model checking, and the their goals are divided into antecedents and consequents (OWRE et al., 2001). Some proof assistants generate proof-objects that store the term to be checked by a simple proof checker, giving greater reliability, which is not the case with PVS, but this fact provides advantages such as allowing all kinds of rewriting for numeric as well as symbolic equalities (NAWAZ et al., 2019).

This tool has been shown to be attractive both for academics and industry having been used successfully in these environments and with several applications, such as checking file systems, analyzing distributed cognition systems, cryptographic protocolos, etc. NASA has also been providing some contributions to support the PVS user community, some of which are: i) NASA PVS Library, a large collection of PVS developments, ii) ProofLite, a Batch prover and proof scripting, iii) Manip and Field, an Algebraic manipulation of real-valued expressions, iv) Sturm and Tarski, decision procedures for single- variable polynomials, v) Interval Affine Arithmetic and Bernstein Polynomials, semi-decision procedures for real-valued expressions (MUÑOZ, 2017).

PVS provides an interactive environment for writing specifications. These PVS specifications are organized from a set of theories. In turn, these theories are formed by a series of statements that are used to introduce type names and constants, definitions, variables, axioms and theorems, into the theory. The IMPORTING clause can also be used to allow importing the visible names of other theories. In addition, we have in PVS an internal library called Prelude, which defines a collection of basic theories about logic, functions, predicates, sets, numbers, among others. An example of constructing a theory in PVS is given below in `MyFirstTheory`, with the definition of square and with the lemma and axiom defined from that.

```
MyFirstTheory:
THEORY BEGIN
square(n:nat): nat = n * n
square nondecreasing: LEMMA
FORALL (n:nat) : square(n) >= n
sqrt : [nat-> nat]
axiom sqrt: AXIOM
FORALL (n:nat) : square(sqrt(n)) <= n
AND n < square(sqrt(n)+1)
 END
MyFirstTheory
```

The theory can make use of standard PVS types, such as `MyFirstTheory`, which uses `nat`

in its definitions. However, type declarations can be used to introduce new type names to the context. These types can be uninterpreted, uninterpreted subtype, interpreted or enumeration type (OWRE et al., 2001). Uninterpreted types are those that do not provide concrete information, imposing the lowest possible level of restriction on the specification, supporting abstraction. Here we give the example of the CK declaration as being an uninterpreted type. At another time, CK is instantiated as a set of items. In PVS, for each instantiation of uninterpreted types, it is necessary to prove that all the assumptions made in the imported theory are also valid for the most concrete theory. Therefore, valid properties for an uninterpreted type of CK must also be proven for its instantiations.

```
CK: TYPE
```

In addition to uninterpreted types, a predicate subtype of a given type can be defined as the subset of elements in a type that satisfies a given predicate. For example, we have the subtype of reals other than zero, which is written as x: real | x /= 0. Despite the simplification that the use of the subtype brings, the type verification is undecidable and can lead to proof obligations called type correction conditions (TCCs). Such TCCs must be proven by the user with the help of the PVS itself.

Finally, among other things, PVS also contains a theorem prover, which provides specific commands for solving each of proof goals. These commands are called rules or strategies. PVS uses the sequential-style proof representation to display the purpose of the current proof goal for the ongoing proof. We give the example of a simple implication A-> B V A. The goals of proofs are divided into formulas known as antecedents (negative numbers) and consequents (positive numbers). The users enter proof commands after “Rule?” and awaits feedback from the tool. In this case, the `flatten` command applies a series of disjunction rules that are sufficient to resolve the implication.

```
{1} A IMPLIES (B OR A)
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
Q.E.D.
```

Another example is the `prop` rule, which generates subgoal sequents when applied to a sequent that is not propositionally valid. After that, the user will have to enter other commands to proceed with this proof.

```
{1} ((A IMPLIES B) IMPLIES A) IMPLIES (B AND A)
Rule? (prop)
Applying propositional simplification,
this yields 2 subgoals:
```

```
{-1} A
|-------
{1} B
```

3.4 THE COQ PROOF ASSISTANT

The Coq proof assistant is a software tool that implements the CIC, presented in Section 3.4.1. The system was designed to develop mathematical proofs, write programs, and has the power to express desired properties of these programs, that is, to define specifications, and develop a set of theories. This is possible due to the two languages used in this tool: Gallina, Vernacular and Ltac. Gallina is the language that allows the development of theories that are constructed from axioms, hypotheses, parameters, lemmas, theorems, definitions of constants, functions, predicates and sets (TEAM, 2017). Vernacular, in turn, is the language of commands, used to define objects, build proof scripts and provide interaction with the user. Finally, Ltac is the tactic language available in Coq.

Coq is one of the most successful proof assistants, and has been growing in popularity among academics, researchers, mathematicians and engineers. The ACM page for the award (Programming Languages Software Award, 2013) which recognizes the development of software systems that have had a significant impact on research, implementations and programming language tools, tells us the following about this system:

"The Coq proof assistant provides a rich environment for interactive development of machine checked formal reasoning. Coq is having a profound impact on research on programming languages and systems, making it possible to extend foundational approaches to unprecedented levels of scale and confidence, and transfer them to realistic programming languages and tools. It has been widely adopted as a research tool by the programming language research community, as evidenced by the many papers at SIGPLAN conferences whose results have been developed and/or verified in Coq. It has also rapidly become one of the leading tools of choice for teaching the foundations of programming languages, with courses offered by many leading universities and a growing number of books emerging to support them. Last but not least, these successes have helped to spark a wave of widespread interest in dependent type theory, the richly expressive core logic on which Coq is based."

Its dependent type feature enables the user to associate types with values in order to build safer and more efficient programs, providing greater control over the data used in these programs. In addition, it is an easy-to-check Kernel proof language that meets the De Bruijn criterion. By this, we mean that the proof assistant creates an 'independently checkable proof object' while the user is interactively proving a theorem (GEUVERS, 2009). So, we can ignore the possibility of errors during the construction of a proof and rely only on a (relatively small) proof verification kernel (CHLIPALA, 2013). This Kernel is implemented using the

OCaml language, and it is highly unlikely that an OCaml bug will interrupt Coq consistency without interrupting all other types of resources from Coq or other software compiled with OCaml (TEAM, 2017).

3.4.1 THE CALCULUS OF INDUCTIVE CONSTRUCTIONS

As for the proof assistants, the issue of the validity of the proof quickly arises. It is important to choose well the formalism that will be used in these systems, not only to avoid failures, but also so that the formalism is expressed in an understandable way. TYPE THEORY is fundamental in this regard. It is concerned with the classification of entities in sets called types and is used by programs to find simple mistakes at compile time, to generate information about data to be used at runtime, as well as in proving theorems, in the study of the foundations of mathematics, proof theory and language theory (NEDERPELT; GEUVERS, 2014).

The formalism implemented by Coq is called the CALCULUS OF INDUCTIVE CONSTRUCTIONS (CIC), and it is an expressive type theory that provides a natural representation of notions like reachability and operational semantics defined through inference rules (PAULIN-MOHRING, 2015). The language behind CIC is CALCULUS OF CONSTRUCTIONS (CC). However, as Coq is designed to be used by an interface in text mode, there are differences in the syntax used for that language.

In CIC, expressions are terms that are associated with types, and types are also associated with types, called classifications, being types and classifications terms. Coq's set of types *S* can be: i) **SProp**, the universe of definitionally irrelevant propositions; ii) **Prop**, the universe of logical propositions; iii) **Set**, the universe of program types or specifications; iv) **Type(i)**, a universe hierarchy with i ≥ 1. So, formally we have:

- S = SProp, Prop, Set, Type (i) | i ∈ N

It is not necessary to define the index i when referring to the universe *Type(i)* because the system itself generates an index for each instance of Type (TEAM, 2017).

The Check command checks whether the expression is well formed and asks Coq to show its type.

```
Check bool.
=> bool: Set
Check 3.
=> 3: nat
Check nat.
=> nat: Set.
```

That said, we have the bool object, for instance, as a predefined type for Boolean values, having the type Set. The constant 3 is the natural type, whose type is in Set. Functions like andb itself are also data values, like true and false.

```
Check andb.
=>  andb
    : bool -> bool -> bool.
```

Propositions have the type Prop, for example, 1 <10 is the type of all proofs that 1 is less than 10 and is also a term of the type Prop. Actually, in general, the convention is that the programs are in Set and the proofs are in Prop. Taking the example of the axioms, they must be of the type Prop and not Set, since the axioms belonging to Prop are always erased by extraction, so we sidestep the axiom's algorithmic consequences.

```
Check 3 =3.
=> 3 = 3
     : Prop.
```

When using the tool, the Coq user may need to deal with universe inconsistency error messages, due to variables that are not in the same type hierarchy, which shows that the type is predicative. This concept is related to the unpredictability, involving inconsistent constructions like "the set of all sets that do not contain themselves"(CHLIPALA, 2013). With regard to Prop in Coq, what we really care about is the provability of a proposition, not the proof of it. There are several ways to prove a proposition, but they are equal in the sense that they provide the same proposition. So their distinction is just irrelevant.

In type theories, there are typing rules that define whether the terms are well typed. These rules depend on E and Γ, which represent, respectively, the *Environment* and the *Context* for the inference. The environment includes all global name definitions and the context includes all local definitions. The notation [] denotes the empty local context, and by Γ1; Γ2 we mean concatenation of the local context Γ1 and the local context Γ2. We denote Γ::(y: T) as being the local environment enriched with the definition of a local variable y, whose type is T, and E; c: T is the global environment enriched with the declaration of a constant c, whose type is T. We also take Γ::(y:= t: T) as the local environment enriched with the definition of the variable y, whose value is t and type is T, and E; c:= t: T is the global environment enriched with the definition of the constant c, with the value t and type T (TEAM, 2017).

Additionally, the expression E [Γ] ⊢ t: T states the fact that the term t has type T in the global environment E and local context, and WF(E) [Γ] is used to state that the global environment E is well formed and the local context Γ is a valid local context in this global environment. With that, we present below some typing rules whose premises are placed above an horizontal line and the conclusion appears below the same line.

- **W-Empty**:

$$\frac{}{\text{WF}([])[]}$$

- **W-Local-Assum**:

$$\frac{E[\Gamma] \vdash \mathrm{T{:}s} \qquad \mathrm{s} \in \mathrm{S} \qquad \mathrm{x} \notin \Gamma}{\mathrm{WF(E)}[\Gamma :: (\mathrm{x{:}T})]}$$

- **W-Local-Def**:

$$\frac{E[\Gamma] \vdash \mathrm{t{:}T} \qquad \mathrm{x} \notin \Gamma}{\mathrm{WF}(E)[\Gamma :: (\mathrm{x{:=}t{:}T})]}$$

- **W-Global-Assum**:

$$\frac{E[] \vdash \mathrm{T{:}s} \qquad \mathrm{s} \in \mathrm{S} \qquad \mathrm{c} \notin E}{\mathrm{WF}(E; c : T)[]}$$

- **W-Global-Def**:

$$\frac{E[] \vdash \mathrm{t{:}T} \qquad \mathrm{c} \notin E}{\mathrm{WF}(E; c := t : T)[]}$$

These are the rules for well-formedness of local contexts and global environments, either for assumptions (*W-Local-Assum* and *W-Global-Assum*) or for definitions (*W-Local-Def* and *W-Global-Def*). The *W-Global-Def* rule, for example, states that so that a variable c, whose value is t and type is T, is placed in the global environment and this is well-formed, the term t that has type T must be placed in the context E and c must not belong to E.

3.4.2 GALLINA

GALLINA is the core language of Coq. It is used to represent proofs, propositions, terms and types. With that, we can formalize theories and specify programs. Since Coq provides a type checker to verify whether the program meets your specification, Coq allows the development of correct programs and the certification of programs already developed.

```
Inductive vowel: Type :=
 | a: vowel
 | e: vowel
 | i: vowel
 | o: vowel
 | u: vowel.
```

Gallina does not exhibit any syntactic distinction between terms and types, implying that the syntax of terms in its grammar is used to represent both value and type terms. In general, everything after := and before the period, is a term in Gallina, and can be PROP, SET or TYPE. As we have previously mentioned in Section 3.4, Coq allows building types from scratch instead of just using existing data in internal resources (PIERCE et al., 2018). Enumerated types are a simple way to describe finite sets (BERTOT; CASTRAN, 2010), whose members are represented by their constructors.

In the above definition, we have the inductive type vowel, being a Type and its constructors are 'a', 'e', 'i', 'o' and 'u'. When we define an inductive type, Coq automatically includes theorems, so that it is possible to reason and compute enumerated types. VOWEL_IND, for example, is the principle of induction associated with the inductive definition. To view it, we can use the command `Check vowel_ind`, and we will get the following result:

```
vowel_ind
     : forall P : vowel -> Prop,
P a ->
P e ->
P i -> P o -> P u -> forall v : vowel, P v.
```

Here we can notice a universal quantification (forall) regarding a logical property P, followed by nested implications, where each premise is P applied to one of its values, the conclusion is that P is valid for all values of vowel. In summary, it indicates that, to check if a property is valid for all values of vowel, we just need to check if it is valid for each one of them.

Coq also includes the recursion principles VOWEL_REC and VOWEL_RECT, which differ from VOWEL_IND because their initial quantifications manipulate a property whose value is in Set or Type, respectively. With these implicit definitions, we are able to define a function on the vowel type simply by providing the values for each vowel. We can also use VOWEL_REC to create recursive definitions, without the command `Fixpoint`, as is the case with the app2 recursive function, which uses list_rec in its construction.

```
vowel_rec
     : forall P : vowel -> Set,
P a ->
P e ->
P i -> P o -> P u -> forall v : vowel, P v.


vowel_rect
     : forall P : vowel -> Type,
P a ->
P e ->
```

```
P i -> P o -> P u -> forall v : vowel, P v.

list_rec :
forall (A : Set) (P : list A -> Set), P nil ->
(forall (a :A) (l : list A), P l -> P (a :: l)) ->
forall (l : list A), P.

Definition app2 (A:Set) (l1 l2:list A) : list A :=
 list_rec A (fun _=>list A) l2 (fun x xs r=> cons A x r) l1.
```

Gallina also allows to create non-recursive functions, using the command `Definition` in the format `Definition name args: type:= term`, where term is the body of the function that we can build, for instance, from pattern enumeration. The construction `match ... with ...end` is a macro that allows the writing of expressions with pattern matching and case analysis, in which each case consists of a pattern, the arrow "=>" and the result term.

When using pattern enumeration, all constructors of the function's argument type must be considered, which is not seen in the function NEXT_VOWEL. As the letter 'u' is not covered, the following error is displayed: Non exhaustive pattern-matching: no clause found for pattern u.

```
Definition next_vowel (v: vowel) : vowel :=
   match v with
     | a => e
     | e => i
     | i => o
     | o => u
   end.
```

The recursive functions are built using the command `Fixpoint`, where the recursive call must occur only in syntactic subterms of one of its arguments and can be captured through pattern matching.

```
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S n') => evenb n'
  end.
```

The function *evenb* is an example of a recursive function that indicates whether a natural number n is even, where n' is a subterm of n. When calling *evenb*, the patterns will be tested and, depending on the result, will call the function again. For this function, pattern matching is done in its single argument:

1. If it is zero, then the number is even. Since this is a base case, the function will end.
2. If it is 1, then false is returned because it is an odd number. This is another base case.
3. Otherwise, we call the function by passing n-2 as an argument, in order to arrive at the base cases of the function.

Coq ensures that all functions will terminated, so it is required that some argument of the recursive calls is "decreasing" (PIERCE et al., 2018).

3.4.3 TACTICS

In Coq, there are expressions that are only used in the context of proofs and that change the current state of the proof, transforming the goal into subgoals. These expressions are known as tactics. The tactics are used for several purposes and below we give examples of some of them and their description.

Table 1 – EXAMPLES OF TACTICS

Tactic	**Description**
tauto	Tauto implements a decision procedure for intuitionistic propositional calculus to solve goals consisting of tautologies that hold in constructive logic.
intuition	The tactic intuition makes use of auto tactics and takes advantage of the search-tree built by the decision procedure involved in the tactic tauto.
contradiction	The contradiction tactic attempts to find in the current context a hypothesis that is equivalent to an empty inductive type, to the negation of a singleton inductive type, or two contradictory hypotheses.
simpl	This tactic tries to reduce a term to something still readable instead of fully normalizing it.

left/right	Replaces a goal consisting of a P /\ Q disjunction with only P or Q.
split	Replaces a goal consisting of a conjunction P /\ Q with two subgoals P and Q.
generalize	This tactic applies to any goal. It generalizes the conclusion with respect to some term.

3.4.4 HOW TO DEVELOP PROOFS

There are some IDEs that allow the manipulation of vernacular files, so that commands can be executed or undone. CoqIDE, shown in Figure 10, is one of the most used in this regard. We can execute the commands in the window on the left using the buttons at the top of the tool. By doing this, the user can analyze the feedback through the colors: yellow, green and red. CoqIDE highlights in yellow "unsafe" commands such as axiom declarations, and tactics like "give_up". The green highlighted part has been verified by Coq. Beside that, Coq highlights the error in red. Also, the user observes the new subgoals and hypotheses generated in the upper right window, and the outputs of commands displayed in the lower right window, which can also indicate errors. Analyzing this feedback, the user can choose to continue executing new commands, or go back a step and try a different command.

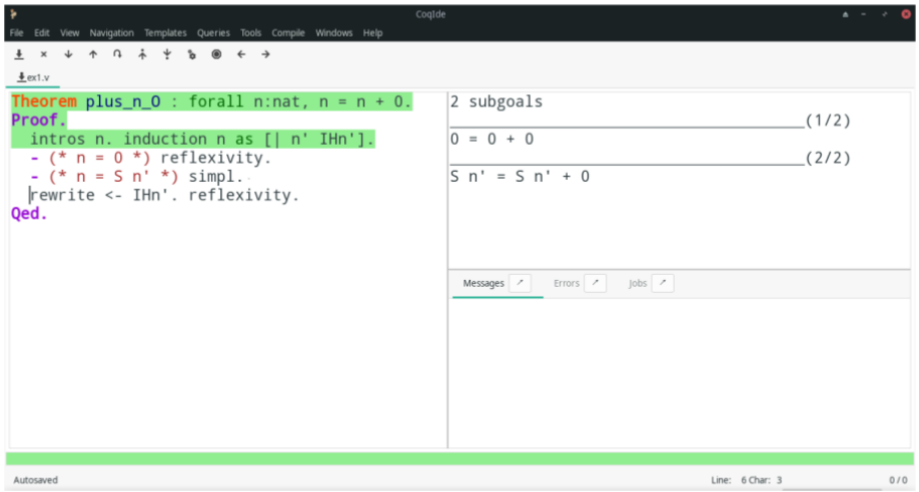

Figure 10 – CoqIDE.

In general, as soon as the user starts to build the proof of a theorem through the command Proof, he applies tactics on the current goal, which can solve that goal, or generate hypotheses and new subgoals in a stack, meaning that they will need to be proven in reverse

order, subgoal by subgoal (YANG; DENG, 2019). Once the stack is empty, the theorem is proved and the command Qed, Save and Defined is used to save it. With `Qed` the name is added to the environment as an opaque constant, but `Saves` saves a completed proof with the name identifier. But `Defined` makes the proof transparent, which means that its content can be explicitly used for type checking and that it can be unfolded in conversion tactics (TEAM, 2017). From this, it is possible to reference the proven sentence and use it through tactics. As illustration, we use the MINUS_N_N theorem which ensures that the subtraction of two equal numbers is zero.

```
Theorem minus_n_n : forall n,
  minus n n = 0.
```

Line	**Tactic**	**Subgoals**

1	intros n.	`1 goal`
		`n: nat`
		`n - n = 0`

2	induction n.	`2 goals`
		`0 - 0 = 0`
		`subgoal 2 is:`
		`S n' - S n' = 0`

3	- simpl.	`1 goal` `0 = 0`
4	reflexivity.	This subproof is complete, but there are some unfocused goals.

5	- simpl. rewrite IHn'.	`1 goal`
		`n': nat`
		`IHn': n' - n' = 0`
		`0 = 0`

6	reflexivity.	No more goals.

Initially, intros are used to introduce variables appearing with forall. The tactic - is used

on lines 3 and 5. These markers are very useful for maintaining the organization of the proofs, in addition to supporting the prover by making visible only the current subgoals, otherwise, Coq shows a list of all subgoals that need to be proved before the entire proof is completed. There are other markers as + and *. The ; tactical applies the tactic on the right side of the semicolon to all the subgoals produced by tactic on the left side.

A successful Coq proof implicitly generates a proof tree. Figure 11 illustrates the proof tree for PLUS_N_0 theorem (forall n : nat, n = n + 0), where the root is the original theorem, whose nodes are goals and the edges are the tactics.

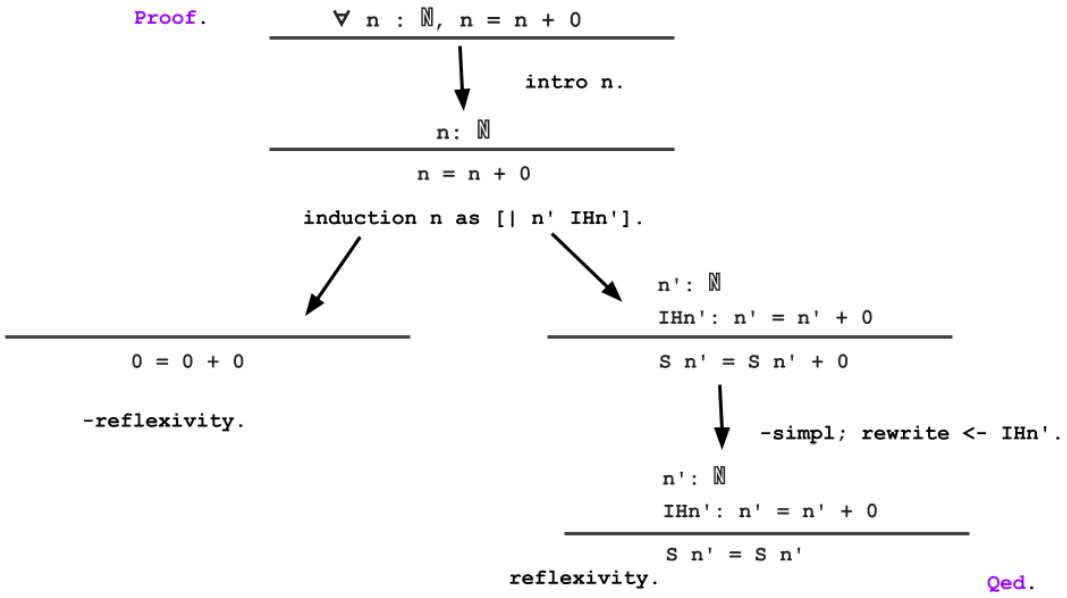

Figure 11 – PROOF TREE.

3.4.5 TYPECLASSES

One of the most important characteristics in high-level languages is polymorphism. Approaches like OOP and TYPECLASSES deal with polymorphism in their developments, but in different ways. Java programmers, for example, confuse classes with Typeclasses. In fact, the latter case is more similar to generic interfaces, which will be composed of functions, but also properties that must be proved by your instances.

Code modularization using Typeclasses has become popular thanks to languages like Haskell and Isabelle, and two possible reasons for using this approach are mathematical clarity and productive code reuse (LUNDY, 2019). Typeclasses and their instances in Coq come with an advantage over these two languages, as they are first class citizens. This means that, in Coq, classes and their instances are implemented as common record types (dictionaries) and registered constants of those types. Although they are records in their essence, Coq uses a particular syntax to declare typeclasses.

```
Class classname (p1 : t1) ...(pn : tn) [: sort] :=
```

```
{ f1 : u1 ; ...; fm : um }.


Instance instancename q1...qm : classname p1...n := {
f1 := t1 ; ...; fm := tm }.
```

Here, `(p1: t1) ... (pn: tn)` are the parameters of the class and `(f1: u1); ...; (fm: um)` are called methods of the class. Furthermore, [: sort] indicates that the type can be Prop, Set or Type. Once we define a class, by instantiating it, we can set these variables with compatible types, functions and existing properties, which include axioms, lemmas and theorems (TEAM, 2017). A typical example of Typeclass is that of the `Eq` class. In addition to the interface, we also present a possible instance.

```
Class Eq A :=
  {
    eqb: A -> A -> bool;
  }.

Program Instance eqBool : Eq bool :=
  {
    eqb := fun (b c : bool) =>
       match b, c with
         | true, true => true
         | true, false => false
         | false, true => false
         | false, false => true
       end
  }.
```

4 COQ FORMALIZATION

Proof assistants can provide formal specifications for algorithms, programs and systems in general. These specifications are composed of types, definitions, functions, lemmas, theorems, their respective proofs, among other elements. The soundness of the theories related to SPL refinement has already been proven through the specification made in the PVS. However, as mentioned in Section 3.1, proofs can play several roles, in addition to showing that some statements are true. Based on this, and taking the attractive facts of the Coq proof assistant, we chose to specify these same theories in Coq. With that, we have the possibility to make a comparison between the two specifications and draw some reflections about the difference between these two systems.

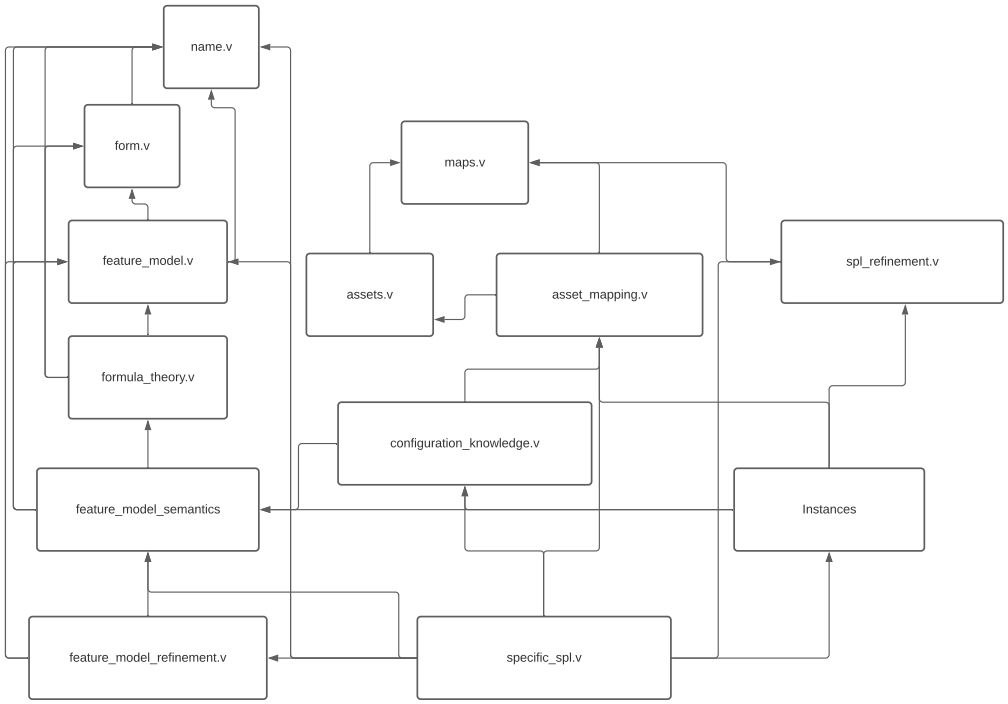

Figure 12 – THEORY STRUCTURE.

In this chapter we present the main results of our work, which is a formalization of SPL Refinement theory, in parallel with the original PVS specification, comparing and contrasting both of them. For brevity, we are omitting some definitions. The complete formalization is available in the project repository.[1]

Figure 12 shows the dependencies between the existing modules, where the arrow means import and Instances are all files referring to Typeclasses. The present formalization can be organized into the following groups:

[1] <https://github.com/spgroup/theory-pl-refinement-coq>

- **Basic definitions**
- **Feature Model**
- **Asset and Asset Mapping**
- **Configuration Knowledge**
- **SPL**
- **Instantiations and Templates**

Regarding the files associated with the basic definitions, FM, Asset and Asset Mapping, Configuration Knowledge, the general theory of SPL, Feature Refinement and that of the templates, we have defined about 39 variables in both systems, 16 types in PVS and 19 in Coq, 60 functions in PVS and 51 in Coq, 3 recursive definitions in PVS and 11 in Coq, 20 lemmas in both systems and approximately 50 theorems.

Additionally, a summary of comparison is presented in Section 4.7. In addition, the proofs and discussion are presented later.

4.1 BASIC DEFINITIONS

A product is described by a valid feature selection, which we call a `Configuration`. Listing 4.1.4 and 4.1.2 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 theory is divided into THEORIES in PVS and MODULES in Coq. 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 it is the case with Configuration.

Listing 4.1.1 – Name and Configuration (Coq)

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

Listing 4.1.2 – 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 formulas for the possible values and relations, such as true/false, feature names, negation, conjunction, and implication, which in PVS comes along with associated ACCESSORS and RECOGNIZERS. The recognizers TRUE?, FALSE?, NAME?, NOT? AND? and IMPLIES? are predicates over the Formula_ datatype that are true when their argument is constructed using the corresponding constructor. The accessors n, f, f_0 and f_1 may be used to extract the arguments.

Listing 4.1.3 – Formula (Coq)

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

Listing 4.1.4 – Formula (PVS)

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

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.

4.2 FEATURE MODEL

Features are used to distinguish SPL products. A feature model is then a set of feature names, with propositional formulae using such names. In Coq, we defined the `names` function, which receives an FM and provides the features, in addition to the `formulas` function, which requires an FM to return a set of formulas, accessing the first and second record fields, respec-

tively. In PVS, names are obtained using set comprehension, despite having access to record fields similar to that of Coq.

Listing 4.2.1 – Names and formulas (Coq)

```
Record FM: Type := {
 features: set Name;
 formulas: set Formula }.
Definition names (fm: FM):=
 fm.(features).
Definition formulas (fm: FM):=
 fm.(formulae).
```

Listing 4.2.2 – Names (PVS)

```
FM: TYPE =
[# features:set[Name],
formulae:set[Formula_] #]
names(fm: FM): set[Name] =
 {n:Name | features(fm)(n)}
```

The names_ function is our first recursive function and it returns the set of features names used in a given formula. To specify a recursive definition in PVS, we need to prove TYPE-CORRECTNESS CONDITIONS (TCCs), to guarantee that the function terminates. This is done by the MEASURE keyword that receives a well-founded order relation, to show that the recursive function is total. Coq employs conservative syntactic criteria to check termination of all recursive definitions, allowing recursive calls only on syntactic subterms of the original primary argument.

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 $\forall x\,y : R, \{x = y\} + \{ x = y\}$, where R is an arbitrary type for a list element. For this reason, we need to introduce axioms such as name_dec to establish that certain types are decidable. Adding axioms is a threat to the validity of this study, but these axioms are mostly limited to types belonging to lists, or to hypotheses we assume to be true from the SPL refinement theory.

Listing 4.2.3 – Names Recursive Function (Coq)

```
Fixpoint names_ (f : Formula) : set Name :=
  match f with
    | TRUE_FORMULA => empty_set Name
    | FALSE_FORMULA => empty_set Name
    | NAME_FORMULA n1 => set_add name_dec n1 nil
    | NOT_FORMULA f1 => names_ f1
    | AND_FORMULA f1 f2 => set_union name_dec (names_ f1)
    (set_diff name_dec (names_ f2) (names_ f1))
    | IMPLIES_FORMULA f1 f2 => set_union name_dec  (names_ f1)
    (set_diff name_dec (names_ f2)(names_ f1))
  end.
```

Listing 4.2.4 – Names Recursive Function (PVS)

```
names(f:Formula_): RECURSIVE set[Name] =
 CASES f OF
  TRUE_FORMULA: emptyset,
  FALSE_FORMULA: emptyset,
  NAME_FORMULA(n1): {n:Name | n1=n},
  NOT_FORMULA(f1): names(f1),
  AND_FORMULA(f1, f2): {n:Name | names(f1)(n) OR names(f2)(n)},
  IMPLIES_FORMULA(f1, f2): {n:Name | names(f1)(n) OR names(f2)(n)}
 ENDCASES
MEASURE complexity(f)
```

In `names_` we also notice that set comprehension are used for defining `NAME_FORMULA`, `AND_FORMULA` and `IMPLIES_FORMULA` constructors in PVS, while in Coq we use the `set_add`, `set_union`, `set_diff` functions of ListSet for the same purpose.

As previously mentioned, a configuration is only considered valid if it satisfies the FM formulae. The `satisfies` function is responsible for capturing this requirement.The traditional rules of propositional logic apply here. For example, a configuration satisfies the conjunction formula $p \wedge q$ if it satisfies p and q. Additionally, and most importantly, a configuration c satisfies the feature name formula n if n is a value in c. We opted for Prop instead of bool in our specification, since we are constantly using ListSet functions that use Prop, and sentence construction must also be in Prop. Otherwise, we would have to do several castings throughout our work.

Listing 4.2.5 – 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.
```

The semantics of FMs are 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.The `satImpConsts` and `satExpConsts` functions support this check. The first one returns True if all the names of a configuration are contained in the FM, and the second function returns True only if for all formula in a given FM, it is evaluated as true for a given configuration. Meanwhile, with declarative style

of specification, PVS allow the definition of sets using set comprehension: A = {x: B | P (x)}, which allows a simplified declaration of this function.

Listing 4.2.6 – FM semantics (Coq)

```
Fixpoint filter (fm:WFM) (s: set Configuration) : set Configuration
   :=
  match s with
   | nil => nil
   | a1 :: x1 =>
        if Is_truePB ((satImpConsts fm a1) /\
              (satExpConsts fm a1)) then a1 :: filter fm x1
         else filter fm x1
  end.

Fixpoint genConf (fm : set Name) : set Configuration :=
  match fm with
  | nil => nil
  | x :: xs => (set_add conf_dec (set_add name_dec x (nil)) (genConf
      xs)) ++ (genConf xs)
  end.

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

Listing 4.2.7 – FM semantics (PVS)

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

We also define a refinement notion for FMs. It states that any configuration in the original FM (ABS) must also be present in the modified FM (CON). In our work, the functions that capture this idea require that exactly the same configuration is present in both. To account for renaming features we also have a more general definition, that does not require the configuration to be exactly the same, but we omit it from the text for brevity.

Listing 4.2.8 – Refine (Coq)

```
Definition refines (abs : FM) (con : FM) : Prop :=
  forall (c : Configuration), set_In c (semantics abs) ->
    set_In c (semantics con).
```

4.2.1 FEATURE MODEL REFINEMENT THEORY

Developers may be interested in adding features, either MANDATORY or OPTIONAL, for example. The definitions of the functions that represent the addition of features are similar, as we see in the Listings 4.2.9 and 4.2.10, which show the functions of adding optional and mandatory features respectively, differing only in terms of the formula that is added to the original FM.

Listing 4.2.9 – Add Optional Node (Coq)

```
Definition addOptionalNode abs con n1 n2: Prop: =
 set_In n1 (names_ abs) /\
 (~set_In n2 (names_ (abs))) /\
 names_ (con) = set_add name_dec n2 (names_ (abs)) /\
 formulas (con) = set_add form_dec
 (IMPLIES_FORMULA (NAME_FORMULA (n2)) (NAME_FORMULA (n1)))
 (formulas (abs)).
```

Listing 4.2.10 – Add Mandatory Node (Coq)

```
Definition addMandatoryNode abs con n1 n2: Prop: =
 set_In n1 (names_ abs) /\
 (~set_In n2 (names_ (abs))) /\
 names_ con = set_add name_dec n2 (names_ abs) /\
 formulas con = set_union form_dec
 (app ((IMPLIES_FORMULA (NAME_FORMULA (n2)) (NAME_FORMULA (n1))))
 ((IMPLIES_FORMULA (NAME_FORMULA (n1) (NAME_FORMULA (n2)))))
 (formulas (abs)).
```

In both functions, when adding a new feature named *n2* as a child of *n1*, first *n1* must be present in the set name of the original FM *abs*, and *n2* is not contained in this set. The *names_* function returns the set of names for a given FM. As a result, we have *n1* added to the set of names of *abs* and a new formula, which expresses the optional and mandatory relationship between *n1* and *n2*, added to the set of *con* formulas.

Listing 4.2.11 – Add Optional Node (Coq)

```
Theorem addOptNode: forall (abs: WFM) (con: WFM) (n1 n2: Name),
  addOptionalNode abs con n1 n2 ->
    ref ref abs /\ wfFM con.
```

Having defined `addOptionalNode` and `addMandatoryNode`, some behaviors expected from adding a feature have been specified and proven. The `addOptNode` theorem provides the guarantee that when adding an optional feature, the original FM is refined by the new one, that is, the set of configuration of *abs* is present in the set of configuration of *con*. Additionally, *con* is well-formed.

4.3 ASSET AND ASSET MAPPING

SPLs have 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, as shown in Listing 4.3.1, which are basically key-value pairs, where S represents the type of all keys and T the type of all values that can be mapped to keys, and both are uninterpreted types.

Listing 4.3.1 – Asset and Asset Mapping (Coq)

```
Inductive S: Type.
Inductive T: Type.
Definition pair_ :Type := prod S T.
Definition map_  :Type := list pair_.

Definition Asset     : Type := Maps.T.
Definition AssetName : Type := Maps.S.
Definition AM := map_.
```

An AM is a unique mapping, where an asset name is related to only one asset. In the Coq specification it was necessary to use the `isMappable` function that indicates whether there is a mapping of a given key and a given value, in a given map `s`, in contrast to the PVS specification in Listing 4.3.3, which uses predicate subtypes.

Listing 4.3.2 – Mapping is unique (Coq)

```
Definition unique s : Prop :=
forall (l: S) (r1 r2: T),
 (isMappable s l r1 /\ isMappable s l r2) -> r1 = r2.
```

Listing 4.3.3 – Mapping is unique (PVS)

```
unique(s): bool =
FORALL(l,r1,r2): (s(l,r1) and s(l,r2) => r1=r2)
```

The SPL refinement notion relies on an asset refinement notion. For generality, we assume `assetRef`, as the theory does not depend on a particular asset language. The basic intuition is that it returns a proposition that is 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 4.3.4 – Asset Refinement (Coq)

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

Listing 4.3.5 – 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, although there are also libraries for that.

Listing 4.3.6 – Refinement is preorder (Coq)

```
Axiom assetRefinement:
 forall x y z:set Asset, assetRef x x /\
    assetRef x y -> assetRef y z ->  assetRef x z.
```

Listing 4.3.7 – Refinement is preorder (PVS)

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

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 4.3.8 – 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 4.3.9 – 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)))
```

4.4 CONFIGURATION KNOWLEDGE

The CK relates features to assets and there are different representations of this model. One of them is the association of feature expressions to the set of asset names. This representation is called compositional and we specify the CK as a list of `items`, defined as a record whose elements are asset names and formulas, which represent features expressions.

Listing 4.4.1 – Configuration Knowledge (Coq)

```
Record Item: Type: = {
  exp_: Formula;
  assets_: set AssetName;
}.

Definition CK: Type: = set Item.
```

The CK semantics is defined as mapping AMs and product configurations to a finite set of assets. The function `eval` is the auxiliary function of `semantics_` that yields only the asset names evaluated as true for a given product configuration, aided by the `assetsCK` function.

Listing 4.4.2 – Semantic of CK (Coq)

```
Fixpoint assetsCK items: set AssetName :=
    match items with
      | nil => nil
      | x :: xs =>  (x.(assets_)) ++ (assetsCK xs)
    end.

Definition eval ck c: set AssetName: =
    assetsCK (evalCK ck c).

 Definition semantics_ (ck: CK) (am: AM) (c: Configuration):
    set Asset:= maps a am (eval ck c).
```

When evolving an SPL, in addition to proving that the original SPL is refined by the new one, we also need to ensure the well-formedness of the target. For that, we need additional restrictions on the CK. In this case, the constraint requires that all feature expressions in CK refer only to FM features, expressed by the third line of `wfCK`, and all features names that appear in CK are in the AM domain, expressed by the first line of this function using `set_diff`, which is the set difference function. For two sets x and y `set_diff` returns the set of all els of x that does not belong to y. Beside that, by the third line of `wfCK`, where `wf` is an auxiliary definition that captures the requirements for `exp` being well-typed with respect to the feature model `FM`.

Listing 4.4.3 – CK is well-formed (Coq)

```
Definition wfCK (fm: WFM) am ck: Prop: =
 forall c, set_In c (semantics fm) ->
  (set_diff an_dec (eval ck c) (dom am) = nil) /\
   forall exp, set_In exp (exps ck) -> wt fm exp.
```

4.5 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 estalish 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 4.5.5.

Listing 4.5.1 – Software product lines (Coq)

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

Listing 4.5.2 – Software product lines (PVS)

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

We are then able to formalize the SPL refinement function, as in Definition 2. According to this function, for all existing configuration in the semantics of the FM of pl1, given by `FMRef`, there must be an equivalent configuration in the semantics of the FM of pl2. Additionally, the set of assets of pl2, given by the semantics of CK (`CKSem`) refines the set of assets of pl1.

Listing 4.5.3 – 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))))).
```

Note that this function does not give us guarantees that a given configuration, which was present in the original SPL, will be present in the resulting SPL. Thus, a stronger notion of refinement is also needed, which is sensitive to the name and semantics associated with each resource. That's what we have in Listing 4.5.4.

Listing 4.5.4 – Stronger SPL refinement (Coq)

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

In practice, it might not be the case that all three elements are changed in an evolution scenario. 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), which is represented by the functions `equivalentFMs` and `equivalentCKs`, the AM is refined as previously described in Listing 4.3.8, 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 4.5.5 – Compositionality (Coq)

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

Listing 4.5.6 – 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 #)
```

Additionally, we specify the weaker compositionality theorem, that uses the weaker CK equivalence notion, which establishes equality according to the FM semantics. This is useful when we intend to support operations such as replacing an equivalent feature expression in the CK, as shown in the templates section. This theorem also makes use of the FM refinement notion, which is represented by the function `FMRefinement`, which indicates whether there

is refinement between two FMs.

Listing 4.5.7 – Weak Compositionality (Coq)

```
Theorem weakFullCompositionality:
 forall (pl: PL) (fm: FM) (am: AM) (ck: CK),
  (FMRefinement (getFM pl) fm) /\ equivalentCKs (getCK pl) ck /\
    wfPL (pl2) -> plRefinement pl
    {| pls :=  ((fm ,(getAM pl)), (getCK pl));
      wfpl := wfPL  ((fm ,(getAM pl)), (getCK pl))}.
```

4.6 THEORY INSTANTIATION AND TEMPLATES

Even though we present concrete FM and CK languages previously, 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. 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 4.6.1 – 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 present the example of the `plStrongSubset` theorem, which states that if there is a stronger refinement between pl1 and pl2, then for all configuration present in the semantics of the pl1 FM, there is this same configuration in the semantics of the pl2 FM. 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.

We use the `Program Instance` keyword to define a 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 4.6.2 – 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;
    strongerPLRefinement : PL -> PL -> Prop;
    ...

    (*===========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)));
    ...
}.
```

Listing 4.6.3 – SPL Instance (Coq)

```
Program Instance Ins_SPL: SPL Asset AssetName AM Conf FM AM CK PL:=
    {
     plRefinement:= plRefinement_func;
     strongerPLRefinement:= strongerPLrefinement_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. Not all properties are fully proven, which is a limitation of our study. We have so far provided eight properties out of fifteen.

4.6.1 REPLACE FEATURE EXPRESSION TEMPLATE

As described in Section 2.5, REPLACE FEATURE EXPRESSION guarantees that it is safe to change a feature expression associated to an asset *n* in the configuration knowledge from *e* to *e'* as long as the restriction that these expressions describe are equivalent considering the FM is respected. When we specify the templates, we define functions that express the similarities and differences between the original and the resulting SPL.

One of these functions is the syntax, which represents the syntactic relationships between *LHS* and *RHS*. From Figure 6, we notice that F and A, which are FM and AM respectively, remain the same, so there is no need to mention them here. In addition, only one line in the CK is divergent between SPLs. This line is expressed as a CK `item`. So, syntactically, we have that the source CK `ck1` corresponds to the union of a given `item1` with `items`, which are the other items that do not change. We also have that the resulting CK `ck2` corresponds to the union of `item2` with `items`, and the set of assets of `item1` and `item2` is the same.

Listing 4.6.4 – Syntax of Replace Feature Expression (Coq)

```
Definition syntaxReplaceFeatureExp ck1 ck2 item1 item2 items: Prop: =
 (ck1 = app items item1) /\
 (ck2 = app items item2) /\
 item1.(assets) = item2.(assets).
```

Another function is conditions, which represents the transformation preconditions so that change is safe. In this case, the constraint is that all configurations of F lead to equivalent evaluation for the feature expressions in both `item1` and `item2`. We also specify that the features expression in `item2` is well-typed with respect to F, that is, any feature referred to in the expression belongs to F.

Listing 4.6.5 – Conditions of Replace Feature Expression (Coq)

```
Definition conditionsReplaceFeatureExp fm item1 item2: Prop: =
 wt fm (item2.(exp)) /\
   forall c, set_In c (semantics fm) ->
     iff (satisfies (item1.(exp)) c) (satisfies (item2.(exp)) c).
```

We then establish the auxiliary theorem `replaceFeatureExp_EqualCKeval`, which uses `syntaxReplaceFeatureExp` and `conditionsReplaceFeatureExp` to ensure that replacing the feature expression by an equivalent one results in the same set of assets when evaluating the CK.

Listing 4.6.6 – Theorem of Replace Feature Expression (Coq)

```
Theorem replaceFeatureExp_EqualCKeval
 {Mp: Maps Asset AssetName AM}
  {Ft: FormulaTheory Formula Name FM}
   {FtM: FeatureModel FM Configuration}
    {AssetM: AssetMapping Asset AssetName AM}
     {ckTrans: CKTrans FM Asset AM CK Configuration}
      {SPL: SPL Asset Configuration FM AM CK PL}:
   forall pl ck2 item1 item2 items,
(
 (
   wfCK (getFM pl) (getAM pl) (getCK pl) /\
   syntaxReplaceFeatureExp (getCK pl) ck2 item1 item2 items /\
   conditionsReplaceFeatureExp (getFM pl) item1 item2
 )
 ->
   forall ( c : Configuration ) , set_In c ( semantics ( getFM pl ) )
   -> semantics_ ( getCK pl ) ( getAM pl ) c
      = semantics_ ck2 ( getAM pl ) c
).
```

Finally, the `replaceFeatureExpression` theorem encodes the soundness proof for this template. The general theorem for proving soundness establishes that we have to prove both SPL refinement and well-formedness of the resulting SPL. The previous lemma establishes that when we replace a feature expression by another which is equivalent, we generate the exact same products. Therefore, this results in a well-formed and refined SPL, due to the above lemma and reflexivity of the asset refinement notion.

Listing 4.6.7 – Theorem of Replace Feature Expression (Coq)

```
Theorem replaceFeatureExpression
 {Mp: Maps Asset AssetName AM}
  {Ft: FormulaTheory Formula Name FM}
   {FtM: FeatureModel FM Configuration}
    {AssetM: AssetMapping Asset AssetName AM}
     {ckTrans: CKTrans FM Asset AM CK Configuration}
      {SPL: SPL Asset Configuration FM AM CK PL}:
   forall pl ck2 item1 item2 items,
(
 (
   wfCK (getFM pl) (getAM pl) (getCK pl) /\
   syntaxReplaceFeatureExp (getCK pl) ck2 item1 item2 items /\
   conditionsReplaceFeatureExp (getFM pl) item1 item2
 )
 ->
   plRefinement pl (gerPL fm2 (getAM pl) ck2) /\
   wfCK (getFM pl) (getAM pl) ck2
).
```

4.6.2 SPLIT ASSET TEMPLATE

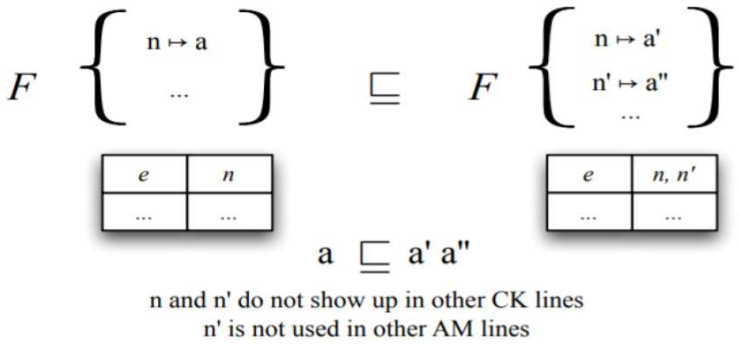

Figure 13 – SPLIT ASSET TEMPLATE (NEVES ET AL., 2015).

The SPLIT ASSET TEMPLATE described in Figure 13 indicates that it is possible to split an asset *a* into two other assets *a'* and *a"*, as long as the composition of the assets *a'* and *a"* refine the asset *a*. This is considered a SPL refinement because for each product that contained the asset before, there is now a corresponding product that contains the composition of the assets *a'* and *a"* after the change. This is useful in situations where you want to modularize code, for example. Figure 14 illustrates this case where the developer wants to extract the

INTERRUPT feature code, which is contained in three different Java classes, and move it to PMINTERRUPTIONASPECT.

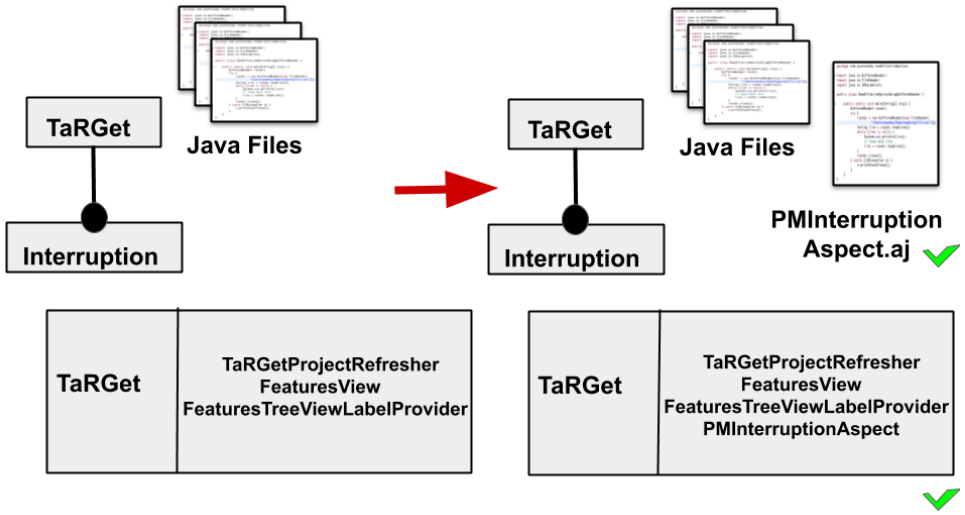

Figure 14 – SPLIT ASSET CASE (NEVES ET AL., 2015).

As for the syntax function, for this template, the first thing to note is that F is invariant, so it doesn't appear in `syntaxSplitAssets`. In this function we also have that the asset mappings `am1` and `am2` differ only in relation to the modified assets, the other lines of AM (`pairs`) are the same. The same occurs with `ck1` and `ck2`, which have the same feature expression, but which differ in terms of `item1` and `item2`, which has an extra asset name, since we created a new asset by dividing the original.

Listing 4.6.8 – Syntax of Split Asset (Coq)

```
Definition syntaxSplitAssets am1 am2 ck1 ck2 item1 item2: Item items
an1 an2 a1 a2 a3 pairs: Prop: =
 am1 = set_add pair_dec (an1, a1) pairs /\
 am2 = set_add pair_dec (an1, a2) (set_add pair_dec (an2, a3) pairs) /\
 ck1 = app item1 items /\
 ck2 = app item2 items /\
 item1.(exp) = item2.(exp) /\
 item1.(assets) = set_add an_dec an1 nil /\
 item2.(assets) = set_add an_dec an1 an2.
```

The definition `conditionsSplitAssets` establishes the precondition that the asset `a1` is refined by `a2` and `a3`. Besides, the remaining CK items cannot refer to `an1` or `an2`.

Listing 4.6.9 – Condition of Split Asset (Coq)

```
Definition conditionsSplitAssets a1 a2 a3 an1 an2 items: Prop: =
    assetRef_func a1 (app a2 a3) /\
      forall item, set_In item items
        -> ~(set_In an1 (item.(assets))) /\
          (~set_In an2 (item.(assets))).
```

To prove this template, we use auxiliary lemmas over the syntax and conditions predicates defined, which we omit here for brevity. First, `splitNotEvalItem` ensures that CK evaluation is not affected if the feature expression of `item1` is not activated, resulting in the same products. This addresses configurations without the asset a. `splitEvalRemainingItems` ensures that evaluating the remaining items in the CK (items) yields the same set of assets in the original and resulting CK. Finally, `splitEvalItemUnion` states that if `item1` is evaluated as true, CK evaluation is equal to the union of the assets mapped by this item — {a} in the original SPL, and {a',a"} in the resulting SPL — with the remaining items.

Based on these auxiliary lemmas, we can prove the `splitAsset` theorem, and ensure that Split Asset is a safe evolution template. When `item1` is not activated, the products are the same, and when it is activated, refinement follows from the compositionality of asset refinement, since {a} ⊑ {a',a"}.

Listing 4.6.10 – Condition of Split Asset (Coq)

```
Theorem splitAsset {Mp: Maps Asset AssetName AM}
 {Ft: FormulaTheory Formula Name FM}
  {FtM: FeatureModel FM Configuration}
   {AssetM: AssetMapping Asset AssetName AM}
    {ckTrans: CKTrans FM Asset AM CK Configuration}
     {SPL: SPL Asset Configuration FM AM CK PL}:
 forall pl am2 ck2 item1 item2 a1 a2 a3 an1 an2 items pairs,
   (
     (
      wfCK (getFM pl) (getAM pl) (getCK pl)  /\
      syntaxSplitAssets (getAM pl) am2 (getCK pl) ck2 item1 item2 items
          an1 an2 a1 a2 a3 pairs /\
      conditionsSplitAssets a1 a2 a3 an1 an2 items
   )
   ->
     plRefinement pl (gerPL (getFM pl) am2 ck2) /\
     wfCK (getFM pl) am2 ck2) /\
     wfPL (gerPL (getFM pl) am2 ck2)
 ).
```

4.7 SUMMARY

In this work, we made a direct mapping of the specification of the SPL refinement theory made in PVS to the proof assistant Coq. In general, both specifications are similar for most encodings. Most of the differences noted are presented earlier in this chapter and their key aspects are summarized in Table 2, which is divided between the existing sections, the type of definition made, and how the specification was made in both systems. For the definition of FM semantics in the FM section, for instance, we have followed an operational specification in Coq and declarative specification in PVS.

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

Coq and PVS provide powerful integrated languages in terms of syntax expressiveness, used to define different structures such as uninterpreted types, records, recursive functions and, as systems used to prove theorems, these proof assistants also provide a means of specifying lemmas and theorems. Figure 15 shows the total number of lines and Figure 16 shows number of specific concepts used in the specifications. The figures presented includes the files associated with the **basic definitions**, **FM**, **Asset** and **Asset Mapping**, **Configuration Knowledge**, the **general theory of SPL**, **Feature Refinement** and that of the **templates**, without taking the proofs into account.

First, we note the difference in the number of lines that indicates the PVS specification as being less verbose. PVS took advantage of the definitions made with subsets, subtypes, of not needing one to use some keywords (such as Inductive, Definition in Coq), etc. In addition, most of the extra lines in Coq are related to the specifications of the templates that require some restrictions to be satisfied. The import numbers are similar, however, nine `Require Import` were used to include libraries in Coq.

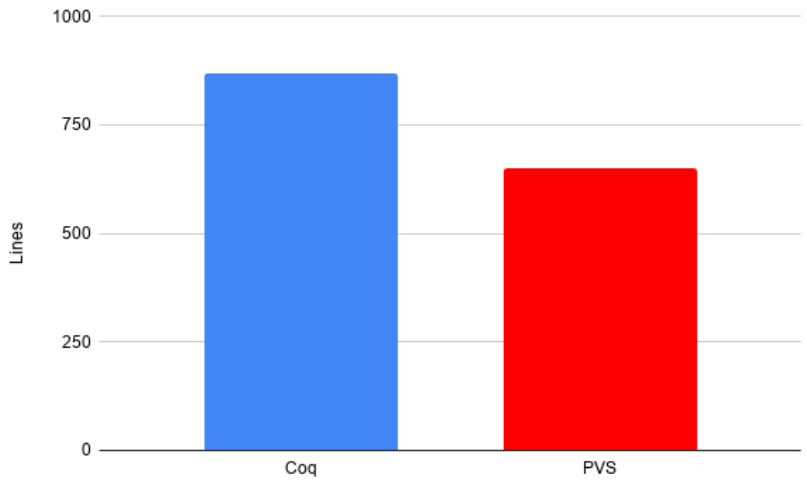

Figure 15 – TOTAL OF LINES IN COQ AND PVS SPECIFICATIONS.

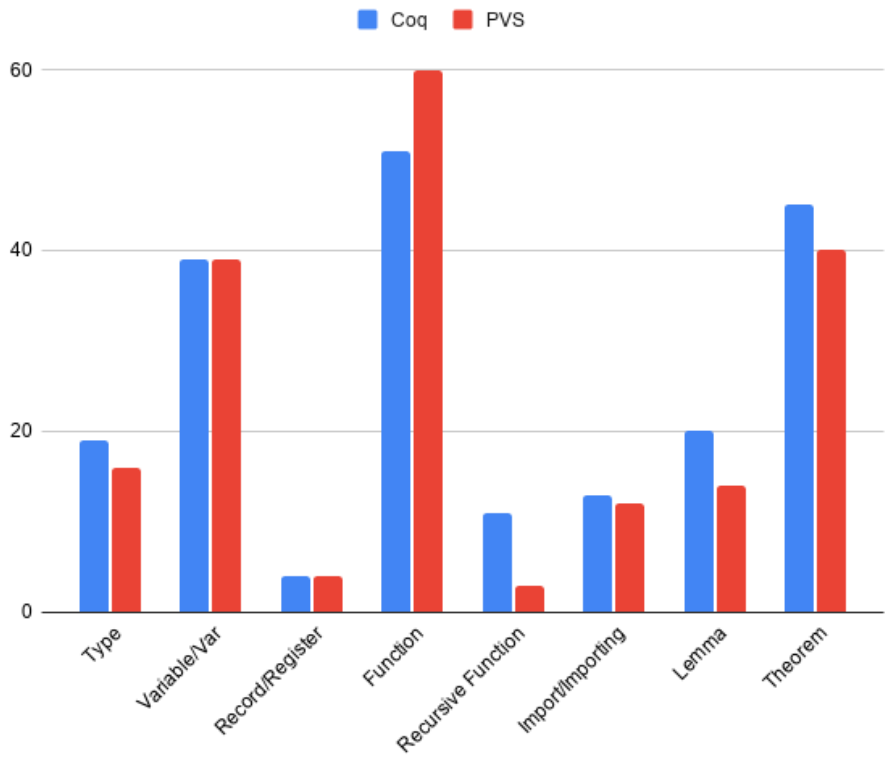

Figure 16 – SUMMARY OF SPECIFICATION DEFINITIONS.

The biggest difference is in the number of functions and recursive functions. Not using Dependent Types in Coq meant that we used a few more definitions to express what had been defined in PVS. An example with extra definitions is the definition of `lemmaSetComp` in Listing 4.7.1 and Listing 4.7.2.

Listing 4.7.1 – Lemma lemmaSetComp (PVS)

```
lemmaSetComp : LEMMA
 FORALL(ck,c):
 {i:Item | ck(i) AND satisfies(exp(i),c) } =
 intersection({i:Item | ck(i)}, {i:Item | satisfies(exp(i),c) })
```

Listing 4.7.2 – Lemma lemmaSetComp (PVS)

```
Fixpoint SetCompAux ck c: set Item :=
    match ck with
      | nil => nil
      | x :: xs => if Is_truePB (satisfies (getExp x) c) then
                   x :: (SetCompAux xs c) else (SetCompAux xs c)
    end.

  Lemma lemmaSetComp: forall ck c,
    SetCompAux ck c = set_inter item_dec ck (SetCompAux ck c).
```

It is necessary to define a recursive structure to define the lemma that is simplified with the use of set comprehension in PVS. FM semantics in Section 4.2 is also a good example of this.

4.8 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. 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 the Hypatheon library.[2]

Although we are porting an existing PVS specification, the proof methods often differ between the two systems. For example, Listings 4.8.1 and 4.8.2 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 by simplifying the hypothesis `HMpb` and by using 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 the considered hint databases. These calls generate subgoals that `auto` tries to solve without error, but limited to five attempts by default, to ensure that the proof search eventually ends. The user 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.

Listing 4.8.1 – 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.
```

[2] <https://github.com/nasa/pvslib>

Listing 4.8.2 – 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 foundation 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.

4.8.1 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.[3]

- **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`;

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

  - **PVS:** `flatten`, `case`, `split`, `induct`.

- **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`, `lia`;
  - **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.

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 3 – TOTAL TACTICS BY CATEGORY

Table 3 presents the numbers for such categories in our proofs. 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 this 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.

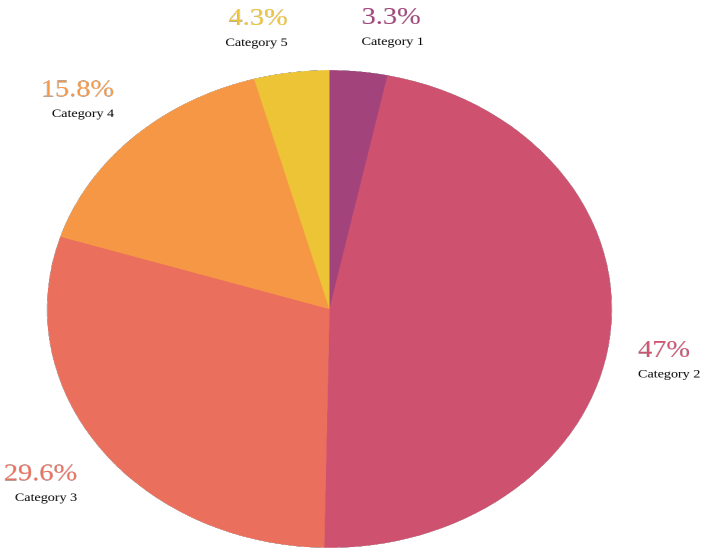

Figure 17 – TACTICS FROM OUR COQ SPECIFICATION

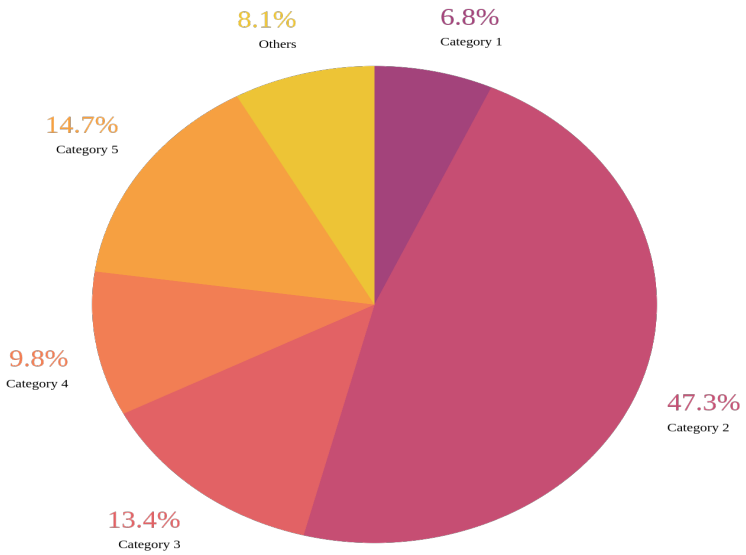

Figure 18 – TACTICS FROM GITHUB

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. For instance, `induction`, which generates from one to six more subgoals in this formalization, was used 15 times in Coq and only 5 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.

Nonetheless, to provide some external validity for our analysis, we also mined Coq repositories from GitHub using GraphQL API v4 and PyDriller (SPADINI; ANICHE; BACCHELLI, 2018). 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 17 and 18 compare the distribution of commands 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.

4.9 DISCUSSION AND LESSONS LEARNED

Coq posed some difficulties during this the development of this work, but has also helped some other tasks. It is important to highlight that participants of this research 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.

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.[4] We are also aware of *The Coq Consortium*,[5] 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.

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 suc-

[4] <https://cstheory.stackexchange.com/questions/tagged/coq>
[5] <https://coq.inria.fr/consortium>

cinct 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 identifiers or by patterns, which might prevent the unnecessary definition of lemmas.

5 CONCLUSIONS

In this work, we port the existing SPL refinement theory mechanized in PVS to Coq. This theory is the basis of previous works (BORBA; TEIXEIRA; GHEYI, 2012; SAMPAIO; BORBA; TEIXEIRA, 2019; NEVES et al., 2015; GOMES et al., 2019) related to safe and partially safe evolution of SPLs, although here we only discuss the safe evolution aspect of this theory. This formalization is initially composed of basic definitions, such as Name, Formula and Configuration. Additionally, having defined an SPL as a triple (FM, AM and CK), we formalized the concrete theories of these models, as well as the specification of the general SPL refinement theory. Typeclasses were used to instantiate these theories, defining the interfaces and proving the obligations generated by their instances. The concept of safe, partially safe evolution and SPL refinement allows the derivation of templates that support developers in SPL evolution scenarios. These templates were also formalized in Coq and most of their theorems have been proven.

We also compared Coq and PVS using the developed specifications, showing the differences observed in the specifications of the corresponding modules, file-by-file. A comparison of the proofs was also made. In this case, we divided the tactics and rules into five categories: Proving Simple Goals, Transforming goals or hypotheses, Breaking apart goals or hypotheses, Managing the local context and Powerful Automatic Commands. For each of the categories, we surveyed the number of commands used in each of the proof assistants and give some reasons for the results found. This categorization was also used for an external evaluation. We mined more than 1900 public projects available on GitHub and compared them with our results.

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. In this case, we agreed that there was less effort regarding the specification of the SPL refinement theory made in this system. The use of subtypes and predicate subtypes simplified several definitions, requiring a smaller number of recursive definitions compared to Coq, for instance, which is important, since PVS has disadvantages regarding this type of definition.

The proofs followed different ways in the two systems, using different methods to solve them. We noticed a greater difficulty in Coq in the proofs that relate to recursive definitions. When using the unfold tactic, for instance, there was a greater difficulty in using simplification tactics and application of other previously proven properties. The proofs of the properties of the general theory of SPL got bigger in PVS. For the *amRefCompositional* proof, the biggest proof, about 121 proof rules were used in PVS, while we used 43 tactics in Coq to complete that same proof. In fact, the proofs of the properties of the general theory of SPL were shorter in Coq than in PVS. However, in general, as we saw in the comparison of

the proof commands, proof rules in PVS reduced the proof effort by the user, having proofs like *wtFormRefinement* that were solved with automatic commands. However, 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 in both definitions and proofs. Its larger community and documentation availability might provide greater support for this purpose. In addition, we must take the constant improvements made to these proof systems into account.

5.1 THREATS TO VALIDITY

As any case study, our work also presents threads to validity. This section discusses these threats in what follows.

5.1.1 EXTERNAL VALIDITY

The first threats to be mentioned in this section concern the generalization of our findings. This study involves a case study, comparing two specifications of the SPL refinement theory and from our results we cannot say that they are representative for all specifications. Despite this, we consider this theory an option to trace this type of comparison, given the variety of definitions that it involves, such as interpreted types, uninterpreted types, enumerated types, recursive functions, records, theorems, lemmas, axioms, among others. The specification also includes the formalization of abstract, concrete entities, instances, in addition to various properties that needed to be proven. In addition, we mine several GitHub repositories in order to give some external validity to our proofs comparison.

5.1.2 INTERNAL VALIDITY

Besides, as mentioned earlier, the fact that the researchers involved in this study have more experience with PVS affects our results, being a threat to the validity of our comparison. With that, we recognize that in our Coq specification there may be better alternatives to existing structures, notations, or libraries that could be used to lessen the effort of this task. We also recognize that there may be tactics that could have been used to simplify the evidence. However, when porting the PVS theory to Coq, most definitions are similar, using similar structures.

5.1.3 CONSTRUCT VALIDITY

In this work, we consider the concept of safe evolution, which requires the preservation of the behavior of all SPL products, when making certain changes. However, this is not suitable for all possible evolution scenarios. For instance, if the developer wants to make a bug fix or remove a feature, there is no way to guarantee that all products have the same behavior. With this we have the notion of partially safe evolution, to support developers in these and

other non refinement situations, where only a subset of the products will preserve their behavior. Because of this, the main limitation of this dissertation is that it does not include the concept of partially safe evolution. Furthermore, there are admitted proofs in *SpecificSPL*, the file containing the templates specification. It could be that some properties which have been assumed, but not proven, may have specification errors. Ideally, all proofs should be completed.

5.2 RELATED WORK

There are several formalizations in Coq. Gonthier et al. presents a large and complete formalization of a proof of the Feit-Thompson Odd Order Theorem in the Coq proof assistant, which took about six years to complete (GONTHIER et al., 2013). Benzaken et al. (BENZAKEN; CONTEJEAN; DUMBRAVA, 2014) presented a specification of the relational model as a first step towards verifying relational database management systems with the Coq proof assistant. Ramos and Queiroz (RAMOS et al., 2015) formalized parts of context-free language theory in the Coq proof assistant, which includes the formalization of closure properties for context-free grammars (under union, concatenation and closure) and the formalization of grammar simplifications. In addition to these, there are works that provide comparisons between proof assistants, as described below.

Wiedijk (WIEDIJK, 2003) 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 p 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 p 2 or by an arbitrary prime number, in addition to verifying whether the users proved the statement using their system's library.

As a result of this comparison, Otter, HOL and Omega presented the lowest formalizations, using 17, 29 and 38 lines, respectively. Most made use of their libraries only, except Otter and Theorem who specified too many lemmas to complete their tests. The Agda system does not have a library, but the formalization did not use any unproved statements. As for PVS and Coq, Coq needed fewer lines to perform its proof, 68 lines versus 77 in PVS.

The study also compared prover theorems regarding the criteria of logic used, framework, dependent type and De Bruijn criterion for showing how 'mathematical' the logic of the system is. It also used the criteria of automated interaction style, Poincaré principle, user automation and powerful built-in automation for shows how much automation the system offers. As a result, the authors put Metamath and Agda as more mathematicians and ACL2 and PVS as more automatic. Despite losing to PVS for automation, Coq is well placed on his level of mathematical logic.

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, specifi-

cally. 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.

Nawaz et al. compares 40 theorem provers in order to provide details on implementation architecture, logic and calculus used, library support, level of automation, programming paradigm, programming language, differences and application areas (NAWAZ et al., 2019) . The authors created research questions to be answered by experts in these theorem provers to obtain relevant information about these systems. This work differs from ours because it does not intend to compare the tools in a practical way, with proof problems for example, but focuses on providing a quick and easy guide to the interested users into the area of theorem provers.

Bodeveix et al. (BODEVEIX; FILALI; MUNÕZ, 1999) formalized the B-Method using Coq and PVS. The B-Method is a uniform language, which is Abstract Machine Notation, to specify, design and implement systems. Its usual development involves an abstract specification, followed by some refinement steps. The main components of the B-Method are: i) Abstract machine, where the project's goal is specified; 2) Refinement, where the specification is clarified to make the abstract machine more concrete; 3) The implementation, where refinement continues, until a deterministic version is reached.

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 in this case, the authors prefer to show the specification in PVS when it differs from that of Coq, since the PVS syntax is closer to B. In addition, the article only presents six definitions, where only three present differences between the specification of PVS and Coq. 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.

Results similar to ours are found in the specification of Invariants, which are introduced as a predicate of Inv restriction over state space. This restriction in PVS is introduced using the predicate subtype of its underlying type theory, which leads to undecidability and, consequently, the PVS type checker produces proof obligations to verify the subtyping. In Coq, a record was used to provide proof of the state space constraint. This is similar to the alternative taken in Coq for the PL specification, in Section 4.5.

In general, the authors presented the following conclusions regarding their comparison:

- Dependent types and subtypes simplify the specification of abstract machines. Coq does not have a subtyping mechanism, but its dependent type theory is more powerful than that of PVS, with respect to type constructors.

- PVS obligations correspond to the concept of proof obligations B. In Coq, theorems are always stated explicitly.

- PVS made use of the WITH clause to represent updates to copied data structures. Coq does not have this type of construction, but Coq's grammar extension features should allow the definition of the construct.

5.3 FUTURE WORK

First, as a future work, we intend to deal with some threats to validity of this work. Starting by continuing the proofs of the SPL safe evolution templates. We provide eight properties of fifteen, proving two templates altogether. In addition, we intend to extend our formalization to consider partially safe evolution as formalized through partial SPL refinement.

We previously presented the limitation of our study regarding the difference in experience in PVS and Coq by the participants of this study. Therefore, we also intend to collect feedback from Coq experts to improve our formalization. We have already been notified of how we can reduce some definitions with the use of advanced notations, fold and Dependent Types, for instance.

Additionally, we plan to address simplification of proofs, taking important Coq features presented in Section 4.9 into account, which was not our focus in this work. In this sense, we already know that Coq has a language called Ltac (and its experimental version Ltac2) that allows the definition of more complex tactics and decision procedures. We also intend to obtain gains with the use of tacticals, which will allow the simplification of our proofs in Coq. The findings regarding the Hint command are also indicative that we can reduce the proof effort with the use of that specific command.

We also plan to conduct a semi-structured interview with experts from the two proof assistants in order to investigate their experiences using these systems. In this interview, we can obtain reports of such experiences, questioning the participants about facilities and difficulties encountered during specifications and tests made. This will also be an opportunity to contrast with the results of this work.

Furthermore, we intend to provide a table, raising the number of various criteria to compare PVS and Coq, such as Prelude, specification evolution, evidence evolution, instantiation, expressiveness, automation, recursion, datatypes, time, support tactics, among others. Textbooks will also be part of this list of criteria, as they provide greater support to users who want to get started with the tools.

REFERENCES

APEL, S.; BATORY, D.; KSTNER, C.; SAAKE, G. *Feature-Oriented Software Product Lines: Concepts and Implementation (1st. ed.)*. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 3642375200.

BENZAKEN, V.; CONTEJEAN, É.; DUMBRAVA, S. A Coq Formalization of the Relational Data Model. In: SHAO, Z. (Ed.). *ESOP - 23rd European Symposium on Programming*. Grenoble, France: Springer, 2014. (Lecture Notes in Computer Science). Disponível em: <https://hal.inria.fr/hal-00924156>.

BERTOT, Y.; CASTRAN, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642058809.

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*. [S.l.]: Springer-Verlag, 1999. p. 32–48.

BORBA, P.; TEIXEIRA, L.; GHEYI, R. A theory of software product line refinement. *Theoretical Computer Science*, v. 455, p. 2 – 30, 2012. ISSN 0304-3975. International Colloquium on Theoretical Aspects of Computing 2010. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0304397512000679>.

BüRDEK, J.; KEHRER, T.; LOCHAU, M.; REULING, D.; KELTER, U.; SCHüRR, A. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, Gesellschaft für Informatik e.V., Bonn, v. 23, p. 67, 10 2015.

CHETALI, B. About the world-first CC smart card certificate with EAL7 formal assurances. 09 2008.

CHLIPALA, A. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. I–XII, 1–424 p. ISBN 978-0-262-02665-9. Disponível em: <http://mitpress.mit.edu/books/certified-programming-dependent-types>.

CLEMENTS, P. C.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. [S.l.]: Addison-Wesley Professional, 2001. (SEI Series in Software Engineering).

CRAIG, W. L. *Reasonable faith : Christian truth and apologetics*. 1st. ed. USA: Wheaton, Ill : Crossway Books, 1994., 1994.

CROW, J.; OWRE, S.; RUSHBY, J.; SHANKAR, N.; SRIVAS, M. A tutorial introduction to PVS, presented at WIFT'95. In: *Workshop on Industrial-Strength Formal Specification Techniques, Boca*. [s.n.], 1995. Disponível em: <http://www.csl.sri.com/papers/wift-tutorial/>.

GEUVERS, H. Proof assistants: History, ideas and future. *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, Springer, v. 34, n. 1, p. 3–25, 2009. ISSN 0256-2499.

GOMES, K.; TEIXEIRA, L.; ALVES, T.; RIBEIRO, M.; GHEYI, R. Characterizing safe and partially safe evolution scenarios in product lines: An empirical study. In: *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: Association for Computing Machinery, 2019. (VAMOS '19). ISBN 9781450366489. Disponível em: <https://doi.org/10.1145/3302333.3302346>.

GONTHIER, G. Advances in the formalization of the odd order theorem. Springer-Verlag, Berlin, Heidelberg, p. 2, 2011.

GONTHIER, G.; ASPERTI, A.; AVIGAD, J.; BERTOT, Y.; COHEN, C.; GARILLOT, F.; ROUX, S.; MAHBOUBI, A.; O'CONNOR, R.; BIHA, S.; PASCA, I.; RIDEAU, L.; SOLOVYEV, A.; TASSI, E.; THÉRY, L. A machine-checked proof of the odd order theorem. In: *Proceedings of the 4th International Conference on Interactive Theorem Proving*. Berlin, Heidelberg: Springer-Verlag, 2013. (ITP'13), p. 163–179. ISBN 978-3-642-39633-5.

JIMÉNEZ, D.; SÁNCHEZ, A. Formal proof understanding , writing and evaluating proofs. In: . [S.l.]: Universitat Oberta de Catalunya, 2010.

KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Pittsburgh, PA, 1990. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>.

LINDEN, F. Van der; SCHMID, K.; ROMMES, E. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007. ISBN 3642090613.

LUNDY, D. *Type Classes in Coq*. [S.l.], 2019. Disponível em: <https://medium.com/@daltonjlundy/type-classes-in-coq-6bde9ad1f15>.

MERRIAM-WEBSTER. *Definition of proof*. 2011. Disponível em: <https://www.merriam-webster.com/dictionary/proof>.

MUÑOZ, C. A. *PVS in Practice*. [S.l.], 2017. <https://cic.unb.br/~ayala/pvsclass17/lectures/CM-3-inpractice.pdf>.

NAWAZ, M. S.; MALIK, M.; LI, Y.; SUN, M.; LALI, M. I. U. A survey on theorem provers in formal methods. *ArXiv*, abs/1912.03028, 2019.

NEDERPELT, R.; GEUVERS, P. H. *Type Theory and Formal Proof: An Introduction*. 1st. ed. USA: Cambridge University Press, 2014. ISBN 110703650X.

NEVES, L.; BORBA, P.; ALVES, V.; TURNES, L.; TEIXEIRA, L.; SENA, D.; KULESZA, U. Safe evolution templates for software product lines. *Journal of System and Software.*, Elsevier Science Inc., v. 106, n. C, p. 42–58, ago. 2015. ISSN 0164-1212. Disponível em: <https://doi.org/10.1016/j.jss.2015.04.024>.

OWRE, S.; SHANKAR, N.; RUSHBY, J. M.; STRINGER-CALVERT, D. W. J. *PVS Language Reference*. [S.l.], 2001. Version 2.4. Disponível em: <http://pvs.csl.sri.com/doc/pvs-language-reference.pdf>.

PASSOS, L.; TEIXEIRA, L.; DINTZNER, N.; APEL, S.; WASOWSKI, A.; CZARNECKI, K.; BORBA, P.; GUO, J. Coevolution of variability models and related software artifacts: A fresh look at evolution patterns in the linux kernel. *Empirical Software Engineering*, v. 21, 05 2015.

PAULIN-MOHRING, C. Introduction to the Calculus of Inductive Constructions. College Publications, v. 55, jan. 2015. Disponível em: <https://hal.inria.fr/hal-01094195>.

PIERCE, B.; AMORIM, A.; CASINGHINO, C.; GABOARDI, M.; GREENBERG, M. *Software Foundation Volume 1 - Logical Foundation*. [S.l.: s.n.], 2018.

POHL, K.; BöCKLE, G.; LINDEN, F. *Software Product Line Engineering: Foundations, Principles, and Techniques*. [S.l.: s.n.], 2005. ISBN 978-3-540-24372-4.

RAMOS, M.; QUEIROZ, R.; MOREIRA, N.; ALMEIDA, J. Formalization of context-free language theory. *The Bulletin of Symbolic Logic*, v. 25, 10 2015.

SAMPAIO, G.; BORBA, P.; TEIXEIRA, L. Partially safe evolution of software product lines. *Journal of Systems and Software*, v. 155, p. 17 – 42, 2019.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. Pydriller: Python framework for mining software repositories. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (ESEC/FSE 2018), p. 908–911. ISBN 9781450355735. Disponível em: <https://doi.org/10.1145/3236024.3264598>.

SUKHWANI, H.; LOPEZ, J. A.; TRIVEDI, K.; MCGINNIS, I. Software reliability analysis of NASA space flight software: A practical experience. In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. [S.l.: s.n.], 2016. v. 3, p. 386–397.

TABAREAU, N. *Why should we trust proof assistants?* [S.l.], 2020. <https://www.openaccessgovernment.org/proof-assistants>.

TARTLER, R.; KURMUS, A.; HEINLOTH, B.; ROTHBERG, V.; RUPRECHT, A.; DORNEANU, D.; KAPITZA, R.; SCHRöDER-PREIKSCHAT, W.; LOHMANN, D. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In: *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*. USA: USENIX Association, 2012. (HotDep'12), p. 3.

TEAM, T. C. D. *The Coq Proof Assistant Reference Manual*. [S.l.], 2017. <https://coq.inria.fr/distrib/current/refman/>.

VILLIERS, M. de. The role and function of proof in mathematics. *Pythagoras*, v. 24, p. 17–24, 1990.

YANG, K.; DENG, J. Learning to prove theorems via interacting with proof assistants. In: CHAUDHURI, K.; SALAKHUTDINOV, R. (Ed.). *Proceedings of the 36th International Conference on Machine Learning*. Long Beach, California, USA: PMLR, 2019. (Proceedings of Machine Learning Research, v. 97), p. 6984–6994.