Refactoring Product Lines

Vander Alves
Informatics Center - UFPE
vra@cin.ufpe.br

Rohit Gheyi
Informatics Center - UFPE
rg@cin.ufpe.br

Tiago Massoni
Informatics Center - UFPE
tlm@cin.ufpe.br

Uirá Kulesza
PUC-Rio
uira@inf.puc-rio.br

Paulo Borba
Informatics Center - UFPE
phmb@cin.ufpe.br

Carlos Lucena
PUC-Rio
lucena@inf.puc-rio.br

Abstract

Adoption strategies for Software Product Lines (SPL) frequently involve bootstrapping existing products into a SPL and extending an existing SPL to encompass another product. One way to do that is to use program refactorings. However, the traditional notion of refactoring does not handle appropriately feature models (FM), nor transformations involving multiple instances of the same SPL. For instance, it is not desirable to apply a refactoring into a SPL and reduce its configurability. In this paper, we extend the traditional notion of refactoring to an SPL context. Besides refactoring programs, FMs must also be refactored. We present a set of sound refactorings for FMs. We evaluate this extended refactoring definition for SPL in a real case study in the mobile games domain.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering; D.2.13 [*Reusable Software*]: Domain engineering

General Terms Design, Languages

Keywords Refactoring, Software Product Lines, Feature Model

1. Introduction

Adoption strategies for Software Product Lines (SPL) frequently involve bootstrapping existing products into a SPL (*extractive approach*) and extending an existing SPL to encompass another product (*reactive approach*), or their combination [13, 5]. The *proactive approach*, in which SPL design and implementation is accomplished for all products in the foreseeable horizon, may be less frequent in practice than the former approaches due to its incurred high upfront investment and risks. Extractive and reactive approaches can be enacted by the application of *program refactorings*.

However, the traditional definition of program refactoring [16, 9] does not take into account intrinsic characteristics of SPL: feature models (Section 3) and configuration knowledge [7] mapping instances of the feature model (FM) to classes and aspects in the solution space. For instance, refactoring of a SPL may have the undesirable effect of reducing its configurability. Another problem is that the traditional notion of refactoring applies only to a single product rather than to a SPL, thereby not taking into account transformations involving more than one product. Therefore, the standard definition of refactoring needs to be extended for SPLs, taking into account their specific characteristics.

In this paper, we extend the traditional notion of program refactorings for SPLs, in which FMs are refactored, in addition to regular program refactoring. For that, we propose a set of sound feature model refactorings. A FM transformation is a refactoring when the resulting FM improves (maintains or increases) the set of all possible configurations (products) of the initial FM. So, a SPL refactoring not only improves code structure, but also the quality of the FM by maintaining or increasing the SPL configurability in extractive or reactive scenarios, respectively. The main contributions of this paper are the following:

- A new extractive program refactoring for software product lines (Section 2);
- A refactoring notion relating multiple feature models (Section 4);
- A catalog of sound feature model refactorings (Section 4);
- An approach for verifying soundness of software product lines refactorings (Section 5).

We evaluate this extended refactoring definition in extracting a SPL from legacy code in the mobile games domain. In this way, developers not only employ traditional refactorings (accordingly compiling and testing to verify whether type safety and behavior are preserved), but also use the sound catalog of FM refactorings we propose so as to guarantee configurability improvement.

2. Refactoring

In this section, we explain issues that need to be addressed when considering refactoring in the SPL context (Section 2.1). Then we propose an extended definition of refactoring for such context (Section 2.2). Finally, we present a new extractive program refactoring involving multiple programs.

2.1 Issues in Product Line Refactoring

The term refactoring was coined by Opdyke in his thesis [16]. He proposed refactorings as behavior-preserving program transformations in order to support the iterative design of object-oriented application frameworks [16]. The cornerstone of his definition is

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-237-2/06/0010. . . $5.00

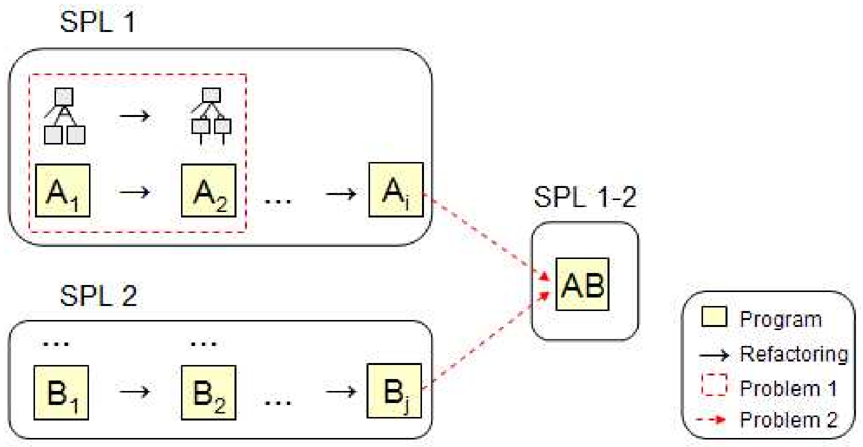


Figure 1. Problems in Refactoring SPLs

that refactorings must maintain correct compilation and observable behavior. In practice, behavior preservation is guaranteed by successive compilation and tests. Opdyke's work and many of the later refactorings apply to frameworks (a technology heavily used today in SPL development) and often introduce variation points. Nevertheless, as *program transformations*, they do not handle configurability-level issues (better addressed at the FM level), nor do they define extractive transformations from two or more existing applications into a SPL.

Figure 1 describes a scenario with two SPLs that are merged into a SPL, where A and B are programs from $SPL1$ and $SPL2$, respectively. In order to accomplish this, refactorings are employed. The $\rightarrow$ arrow represents a refactoring. Figure 1 shows that $SPL1$ and $SPL2$ are refactored to add or expose a set of optional features, as can be seen in their respective FMs (we deliberately omit the FMs of $SPL2$ and $SPL1$-2). Finally, both SPLs are extracted into $SPL1$-2, which addresses all the products configurability. Relying on the standard definition of refactoring, we notice two main issues:

- The definition of refactoring needs to be extended for SPL's context, encompassing configurability improvement by dealing with FMs (Problem 1);
- We need more program refactorings merging multiple programs into one program (Problem 2).

As aforementioned, in practice, a decrement in SPL configurability while refactoring is undesired. However, the traditional notion of refactoring does not take configurability into account, as we can see in Figure 1. If program (source code) A_1 is correctly refactored into A_2, following traditional refactoring steps, still it is not guaranteed that configurability is improved. We must certify that FM_2 (corresponding to program A_2) improves the possible configurations of FM_1. Since the configurability is described by a FM, such model should also be considered during SPL refactoring. So, we need to extend the traditional definition in order to apply FM refactorings (Problem 1).

In order to check configurability improvement, we may rely directly on the semantics of FM to analyze whether the final FM encompasses all the configurations of the initial one. Nevertheless, this may be time-consuming, error-prone, and costly, since analyzing semantics of models may become exponentially hard for large FM models potentially annotated with logical constraints. In order to solve this problem, we propose a catalog of sound FM refactorings that improve configurability and would thus help the developer to evolve FMs (Section 4).

Furthermore, evolving a SPL often involves adapting two or more applications and unifying them, as for extracting products into a SPL. However, this requires program refactorings merging multiple programs into a product line (Problem 2). Traditional refactorings [9] usually transform one program into another. For instance, the traditional refactoring notion is not straightforward in considering a refactoring that merges programs A_i and B_j into the AB program in Figure 1, improving configurability of both SPLs into the new one. In this case, specific program refactorings for SPL are required.

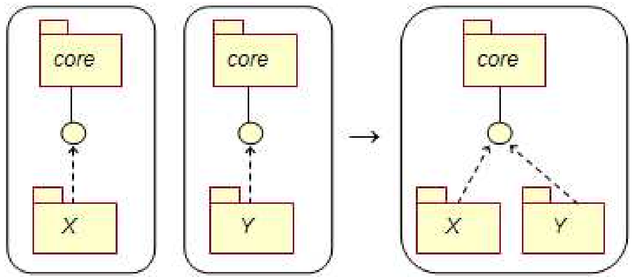
2.2 Product Line Refactoring

In order to deal with Problem 1, we first extend the definition of refactoring for SPLs (in addition to the refactoring catalog for FMs shown in Section 4) :

Definition 1. *SPL refactoring is a change made to the structure of a SPL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products*

In order to deal with Problem 2, we propose a refactoring dealing with several programs. For instance, Refactoring SPL 1 shows an extractive refactoring, in which two existing applications are extracted into a SPL. The refactoring exposes reusable code ($Core$) among the existing applications, thereby removing code duplication. Other code artifacts (X and Y) are kept the same. Each application is now instantiated by reusing asset $Core$. Configurability is not guaranteed to be maintained; for that, feature models must be considered (using the FM refactorings shown in Section 4).

Refactoring SPL 1. ⟨*merge programs*⟩



The SPL refactoring definition consists of three templates: 1) two templates match the code of the existing applications (left side of the arrow); 2) the third template defines how the code of these two applications is extracted into the SPL code. In our approach illustrated in Section 5, Refactoring SPL 1 is used together with traditional program refactorings. In order to guarantee behavior-preservation, compilation and tests are used.

3. Feature Models Overview

A FM represents the common and the variable features of concept instances and the dependencies between the variable features [7]. Each feature model describes, in a tree, a set of features and how they are related.

Relationships between a parent feature and its child features (or subfeatures) are categorized as: *Optional* (features that are optional), *Mandatory* (features that are required), *Or* (one or more must be selected - represented by a filled triangle), and *Alternative* (exactly one subfeature must be selected - represented by a unfilled triangle). Figure 2 depicts these relationships graphically.

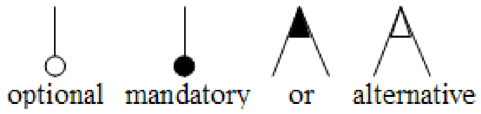


Figure 2. Feature Diagram Notations

Besides these relationships, we allow feature models to include propositional logic formulas about features. For instance, the formula $B \Rightarrow \neg C$ states that if feature B is selected, feature C cannot be selected.

Example. Figure 3 depicts a FM. It has four features (A, B, C and D), one formula ($B \Rightarrow \neg C$) and two relationships: an option relationship between A and B, and an or relationship between A, C and D.

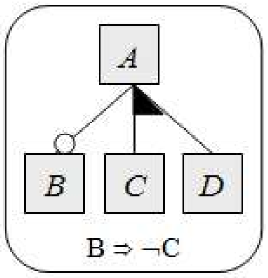


Figure 3. Feature Model Example

The semantics of a FM is the set of its possible (valid) configurations. A configuration contains a set of feature names; if *valid*, it satisfies all constraints of the model. For example, the configurations { A, B, D } and { A, C } are valid for the model in Figure 3. However, the configuration { A, B } is invalid because the or relationship between A, C and D states that whenever A is selected, C or D must be selected.

In order to support the definition of FM refactorings (Section 4), we specified a formal semantics for FMs. Next we show an UML class diagram [4] graphically describing the abstract syntax of this formalization. A feature model contains a set of feature names and a set of formulas. A configuration contains a set of names selected, as specified in Figure 4.

We express all FM relationships in formulas. For example, a mandatory relationship between features A and B is represented by the formula $A \Leftrightarrow B$.

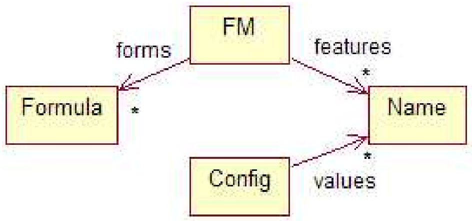


Figure 4. Class Diagram depicting Feature Model Components

Next, we formalize the semantics of a FM, which is given by a set of configurations that satisfy all modeled constraints. The expression `values(c)` yields all selected features in the configuration `c`.

```
semantics(fm:FM): set[Config] =
  {c:Config |
    values(c) ⊆ features(fm) ∧
    ∀ f:forms(fm) | satFormula(f,c)
  }
```

The relation `satFormula` checks whether a configuration satisfies a propositional formula. For instance, considering the model in Figure 3, configuration c satisfies the formula $B \vee C$ if c contains B or C. As another example, a configuration satisfies the formula $B \Rightarrow \neg C$ if c contains B but not C.

4. Feature Model Refactoring

According to Section 2.2, SPL refactoring involves not only program refactoring, but also FM refactoring. Based on this definition, we propose a number of sound unidirectional and bidirectional FM refactorings in Sections 4.3 and 4.4, respectively. Finally, we discuss additional aspects of such refactorings (Section 4.5).

We define FM refactorings as:

Definition 2. *A feature model refactoring is a transformation that improves the quality of a feature model by improving (maintaining or increasing) its configurability.*

Let `m1` and `m2` be two FMs. So, according to Definition 2, `m2` refactors `m1` if and only if all valid configurations of `m1` are valid configurations of `m2`, as formalized next.

```
refactoring(m1,m2: FM): boolean =
  semantics(m1) ⊆ semantics(m2)
```

4.1 Motivation

Figure 5 depicts two small FMs. It describes the colors of a car. In the left-hand side (LHS) FM, a car can be black or white. Suppose that we would like to refactor the LHS model to the right-hand side (RHS) model by adding a new alternative. So we can have an additional blue car in the resulting model, while still maintaining the previous configurations.

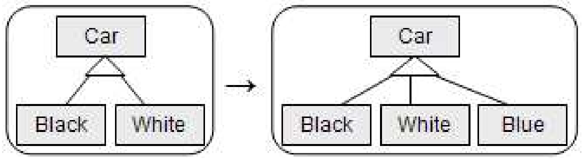


Figure 5. Feature Model Refactoring Example

For ensuring correctness of the refactoring depicted in Figure 5, we have to show that the resulting FM improves the configurability of the initial FM (Definition 2). The LHS FM has two valid configurations: $\{Car, Black\}$ and $\{Car, White\}$. The RHS FM has the same configurations of the LHS FM plus the configuration $\{Car, Blue\}$. Since the RHS model contains all valid configurations of the LHS FM, it is a valid FM refactoring.

Following a similar approach to prove FM refactorings containing considerably more features, relations and formulas may be difficult, time-consuming and error-prone. In order to avoid that, we propose a catalog of sound FM refactorings (Sections 4.3 and 4.4). Next we give an overview of the notation used to state them.

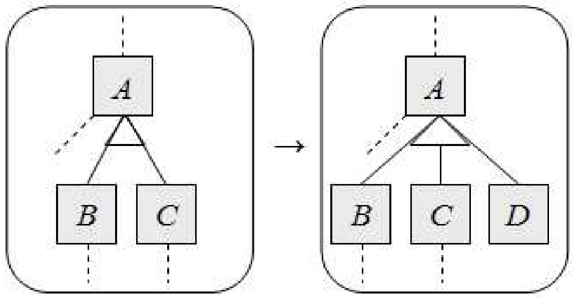
4.2 Refactoring Notation

Each refactoring consists of two templates (patterns) of FMs, on the left-hand (LHS) and right-hand (RHS) sides. We can apply a refactoring whenever the left template is matched by a given FM. A matching is an assignment of all variables occurring in LHS/RHS models to concrete values. Any element not mentioned in both FM templates remains unchanged, so the refactoring templates only show the differences between the FMs. Moreover, a dashed line on top of a feature indicates that this feature may have a parent feature. A dashed line below a feature indicates that this feature may have additional subfeatures.

4.3 Unidirectional Refactorings

Next we propose some FM refactorings in order to solve Problem 1 (Section 2.1). Refactoring 5 allows us to add a new node D and increase the alternative between B, C and D. This refactoring is the general version of the specific refactoring shown in Figure 5. We can apply Refactoring 5 to the specific models depicted in Figure 5 by matching the variables A, B, C and D with the specific features Car, $Black$, $White$ and $Blue$, respectively.

Refactoring 5. *add new alternative*



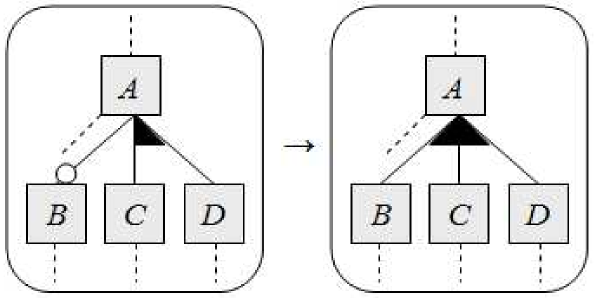
Note that there is no dashed line below D in the RHS of this refactoring because it only introduces a new feature without subfeatures. The other dashed lines of the RHS are necessary to preserve the features matched by dashed lines in the LHS.

Refactoring 5 is sound because the resulting model contains all configurations from the original one, also allowing a configuration containing A and D in the absence of B and C. Therefore, this transformation improves a model by increasing its configurability.

Another general refactoring, Refactoring 2, collapses an optional feature and an or relation into a general or relation encompassing all features. We can propose a similar refactoring for more than two child feature nodes.

Note that Refactoring 2 cannot be applied from right to left because the RHS model can select features A and B (not selecting C and D), which is not possible on the LHS model. Therefore, the resulting model does not contain all valid configurations of the original FM, hence it is not a refactoring. This counter-example illustrates that there are non-trivial configurability-improvement issues in the SPL context, thus further motivating the need for FM refactorings.

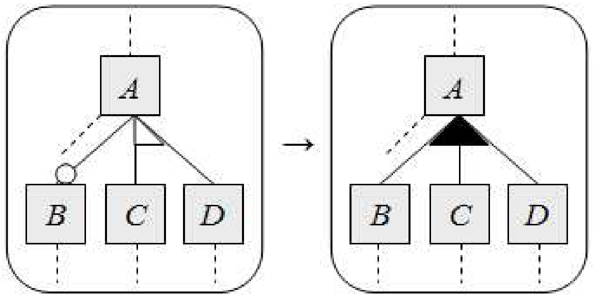
Refactoring 2. *collapse optional and or*



Our catalog of refactorings is summarized in Table 1 and presented in Appendix A. For instance, we have refactorings for pulling up (Refactoring 9) or pushing down (Refactoring 10) feature nodes. Another example, removing a formula (Refactoring 11), is a refactoring since the resulting model is less constrained, hence increasing configurability. Refactoring 12 allows us to introduce an optional feature. In fact, there are additional refactorings, since most of them can be applied similarly in contexts with more than two features, such as Refactorings 2 and 5.

By composing the refactorings, we can derive other valuable refactorings. For instance, by composing Refactorings 1 and 2, we derive Refactoring 3. This is possible because starting from the LHS of Refactoring 3, we can first apply Refactoring 1, thus turning the alternative relationship between C and D into an or-relationship; at this point B is still optional, but we can now apply Refactoring 2 to group B together with C and D into an or-relationship.

Refactoring 3. *collapse optional and alternative to or*



So far we focused on *refactoring single FMs*. However, as we described in Section 2, we may deal with previously existing products or SPLs, each one having its own FM. During extractive evolution, we may want, for instance, to merge these products and SPLs into a single new SPL. In this case, we give support to FMs in the merging refactoring notion presented in Section 2, by defining refactoring between more than two FMs.

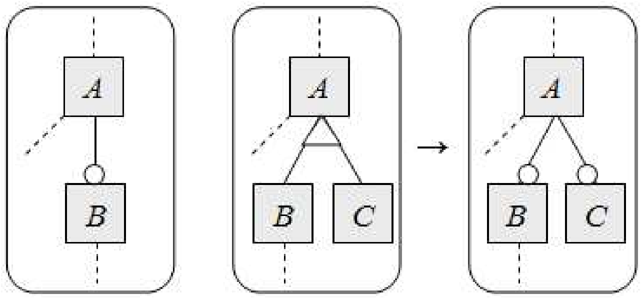
For example, the subsequent refactoring (which we call *Extractive*) allows us to merge optional and alternative relations. The resulting FM refactors the initial ones if and only if the resulting FM refactors each FM on the left side of the arrow. We can actually model an extractive refactoring as a sequence of single-FM refactorings applied to both original FMs separately.

Suppose that $fm1$, $fm2$ represent the two LHS FMs, and $fm3$ the RHS FM, respectively, of the Extractive 1 refactoring. $fm3$ improves the configurations of $fm1$ by applying Refactoring 12 in order to introduce the optional feature node C. Moreover, $fm3$ improves the configurations of $fm2$ by applying Refactoring 8 in

Table 1. Summary of Unidirectional Feature Model Refactorings

Refactoring	Name	Refactoring	Name
1	Convert Alternative to Or	7	Convert Mandatory to Optional
2	Collapse Optional and Or	8	Convert Alternative to Optional
3	Collapse Optional and Alternative to Or	9	Pull Up Node
4	Add Or Between Mandatory	10	Push Down Node
5	Add New Alternative	11	Remove Formula
6	Convert Or to Optional	12	Add Optional Node

Extractive 1. ⟨merge optional and alternative⟩



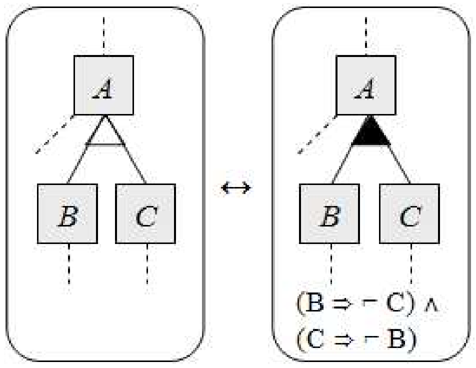
order to convert an alternative to option. Therefore, $fm3$ refactors both $fm1$ and $fm2$. Using the refactorings from Table 1, we can derive other refactorings between more than two FMs.

4.4 Bidirectional Refactorings

A bidirectional refactoring is a *special case* of FM refactoring that *maintains* the configurability of a model. In this section, we propose a set of bidirectional refactorings (B-Refactorings) for FMs. In other words, if two FMs have the same configurability (semantics), we can *always* relate them by applying B-Refactorings.

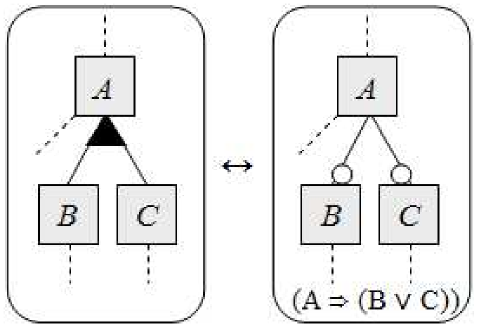
B-Refactorings also define two FM templates, although being applicable in both directions. B-Refactoring 1 relates the alternative and or relations. Applying B-Refactoring 1 from left to right allows us to convert an alternative to an or relation along with two formulas establishing the same constraints. Similarly, by applying the transformation from right to left, we can convert an or to an alternative relation.

B-Refactoring 1. ⟨*replace alternative*⟩



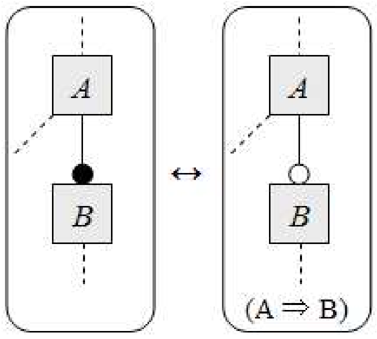
B-Refactoring 2 relates an or relation and optional nodes. Moreover, B-Refactorings 1 and 2 can be applied when there are more than two child features.

B-Refactoring 2. ⟨*replace or*⟩

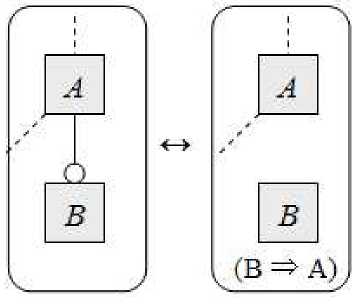


Next, B-Refactoring 3 relates a mandatory feature with an optional feature with a formula stating the same fact, whereas B-Refactoring 4 removes an optional feature and states the same fact in a formula.

B-Refactoring 3. ⟨*replace mandatory*⟩



B-Refactoring 4. ⟨*replace optional*⟩



The root of a FM always appears in all valid configurations. The complete set of B-Refactorings is defined elsewhere [10] and includes one B-refactoring that removes a root and includes a formula stating that the root is always present. Another B-refactoring removes a feature that can never be selected. Similarly, this one allows us to add a set of nodes if we add a formula stating that the nodes cannot be selected. Finally, another B-refactoring allows us to add or remove formulas deducible from the model. Since it is a deducible formula, the configurability is maintained.

Properties of B-Refactorings. Some of the previous transformations may not be useful in practice since they convert a valid FM to another that is not a tree, such as B-Refactoring 4. However, they are important for theoretical reasoning, as we discuss in the following. In practice, developers should only be aware of the FM refactoring catalog.

In this paper we presented only some B-Refactorings. The list of all such refactorings and a proof that they are complete can be found elsewhere [10]. The completeness proof shows that, for any two semantically equivalent FMs, there is a strategy consisting of consecutive applications of B-Refactorings such that it transforms one FM into another. This result means these refactorings are sufficiently expressive for the FM language we consider.

4.5 Discussion

In practice, the developer may choose between *semantics based reasoning* and *reasoning with our catalog of sound refactorings* in order to apply a FM refactoring. Our catalog of sound refactorings can be seen as a high level API, which is much easier to use (based on template matching), whereas semantics based reasoning is similar to using no API at all.

We emphasize the difference between FM refactoring, which we introduce here, and FM specialization, formalized by Czarnecki [8]: FM refactoring is a transformation that either *maintains or increases* the set of all FM configurations, whereas FM specialization is a transformation that *decreases* the set of all FM configurations.

We encoded a semantics for FMs in the Prototype Verification System (PVS) [17], which is a formal specification language. Using the PVS theorem prover, we proved all refactorings proposed with respect to a formal semantics [11]. This experience in PVS was very important for proposing other FM refactorings. Proving them increases the knowledge about other transformations that do not improve configurability. The PVS prover gives us insights of what we have to consider. The formalization and proofs are important in order to increase the reliability when refactoring SPLs, as we present in Section 5.

5. Case Study

In this section, we evaluate the extended refactoring notion for SPLs. First, we describe the context of the case study (Section 5.1); next, in Section 5.2, we describe our approach for verifying the correctness of the case study refactorings in terms of FMs discussed in Sections 4.3 and 4.4.

5.1 Context

Our case study combines the extractive and the reactive SPL adoption strategies [13] in the mobile games domain. J2ME games are mainstream mobile applications of considerable complexity in comparison with other mobile applications [1]. The major variability issues within these products are as follows: optional images, alternative image loading policies, proprietary API, application size limit, screen dimensions, and additional keys [1]. It is essential to note that these features are not independent. Indeed, application size constrains other features, such as optional images and additional keys.

Figure 6 depicts our case study, focusing on the source code. We started from a scenario in which the same game ran in two devices, thus having two initial applications, $Product1$ and $Product2$. Both applications have the same core functionality, but differ in some features, since Device 1 is not a resource-constrained device and, for instance, can afford enough heap and application size for $Product1$ to have the *croma* feature of clouds scrolling in the background and the simple image loading policy of loading all images at game startup. On the other hand, Device 2 is resource-constrained and thus $Product2$ does not have the *croma* feature implemented. Instead, $Product2$ has an optimized image loading policy of loading images on demand during changing screen events.

From this initial scenario, we have two goals: 1) bootstrap the existing products ($Product1$ and $Product2$) into a new SPL (named $SPL1$-2); 2) during this process, react the emerging SPL $SPL2$ to encompass yet another product, which should be the game with a partially (hybrid) optimized image loading policy.

5.2 SPL Refactoring

We apply program and FM refactorings for achieving those goals using our extended definition for SPLs presented in Section 2.2. Although confidence in program refactorings can be increased with a mechanics added by compilation and tests [9], variability improvement is hard to ensure, and tests may not easily uncover such inconsistencies. These problems can usually be detected on the problem space, with FMs. We use the refactorings presented in Section 4.3 for ensuring correctness of FM refactorings by analyzing their application on corresponding FMs after program refactoring steps. We consider that the FM associated with each product is determined from the product documentation or by code examination.

5.2.1 Program Refactoring

In this step, we apply the traditional program refactorings [9] in order to ensure the behavior preservation. In order to accomplish our first goal, we first started applying a sequence of program refactorings to $Product1$ with the aim of modularizing the *croma* and the image loading policy features. In Figure 6, the + symbol in $Product1$ indicates that the implementation of such features is scattered and tangled with the application core. Accordingly, we applied a sequence of object-to-aspect (OA) refactorings (previously introduced by Alves et at [1]) in order to extract such features into aspects *Clouds* and *Startup*, respectively. The result is SPL $SPL1$.

We apply OA and object-oriented (OO) refactorings [9, 1] in order to modularize the $OnDemand$ loading image feature of $Product2$. After that, we accomplish our second goal, evolving the product into a new SPL (named $SPL2$) by adding a new kind of image loading policy ($Hybrid$).

Finally, in order to avoid code duplication, our aim is to integrate $SPL1$ and $SPL2$, due to their similar core. As $SPL1$ and $SPL2$ must have exactly the same core for merging both SPLs, we apply adjustment refactorings (such as renaming) to the core of $SPL1$. We can now apply our merge program Refactoring SPL 1, which was presented in Section 2.2, in order to merge $SPL1$ and $SPL2$ into $SPL1$-2. Notice, that only the last step was not proposed in the traditional notion of refactoring [16].

5.2.2 Feature Model Refactoring

Besides dealing with programs, we must ensure that the resulting transformations improve the configurability of the SPL. For that, some configuration knowledge [7] must be used for defining the correspondence between features and components (for instance, classes and aspects). In the example, we adopt a convention in which features may be tangled with core functionality or implemented as separate aspects; their optionality can be implemented by configuration scripts used for building SPL instances. Other configuration knowledge choices may be used likewise.

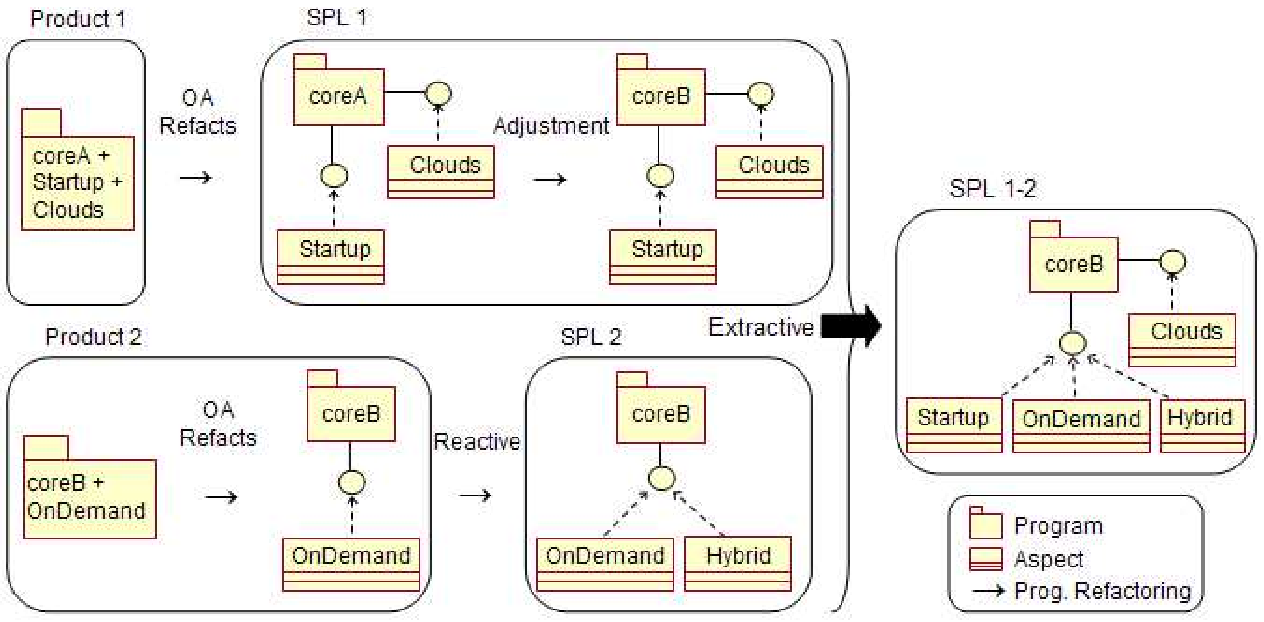


Figure 6. Case Study Program Refactorings

Our approach consists in generating FMs for the initial and resulting product or SPL, investigating the use of the proposed FM refactorings for verifying the configurability improvement between both models. If a sequence of refactorings can be applied, additional confidence on the safety of the SPL refactorings is provided. The original and resulting FMs corresponding to the OA refactoring applied in $Product1$ are shown in Figure 7. For making $Clouds$ optional, we can apply Refactoring 7 resulting into SPL $SPL1$. In a later step, as we applied OO program refactorings to start preparing $SPL1$ for an extractive refactoring, it demands no changes on the FM (a reflexive step).

In the source code for $Product2$, the image loading policy feature was isolated into $OnDemand$ aspect. As $OnDemand$ feature is maintained mandatory, no changes are needed in the FM. At this point in the program, we use the reactive approach for creating $SPL2$, adding the alternative *Hybrid* aspect. Refactoring 5 adds the $Hybrid$ feature, which verifies this step on the FM.

The final step in the source code was an extractive refactoring merging $SPL1$ and $SPL2$. The resulting SPL ($SPL1$-2) thus encompasses $SPL1$ and $SPL2$. At the FM level, we can generate $SPL1$-2 (Figure 7), which includes the three alternative features for image loading ($Startup$, $OnDemand$ and $Hybrid$) and an optional feature ($Clouds$). With the two intermediate FMs for $SPL1$ and $SPL2$, we can now generate a single FM, based on the definition of extractive refactoring given in Section 4.3. We ensure correctness by applying refactorings to both FMs, as shown in the statement below.

$SPL1 \rightarrow SPL1\text{-}2$ $[by\ applying\ 2x\ Refactoring\ 5]$
$SPL2 \rightarrow SPL1\text{-}2$ $[by\ applying\ Refactoring\ 5\ and\ 12]$

This approach shows the application of FM refactorings in a real scenario. Besides ensuring confidence on correct transformations, this approach may also help identifying incorrect steps in a refactoring application, usually not easily detected when inspecting or testing source code. The impossibility of applying certain refactorings may be a consequence of such errors, more easily uncovered at the FM level. It must be stressed that, although some synchronism between FMs and programs with code is important, it is not the focus of this paper. Rather than *monitoring* source code refactoring, the paper proposes a *complementary* transformation level: FM refactorings, for aiding refactoring soundness for SPLs. More ambitious accomplishments, such as, a completely model-driven approach to SPL refactoring, require a formal notion of conformance between FMs and programs, which is regarded as future work.

6. Related Work

A related approach proposes [14] Feature Oriented Refactoring (FOR), which is the process of decomposing a program, usually legacy, into features. Such work focuses on configuration knowledge, specifying the relationships between features and their implementing modules, backed by a solid theory. Also, the authors present a semi-automatic refactoring methodology to enable the decomposition of a program into features. However, FOR focuses so far on bootstrapping a SPL from an *existing* application, rather than two or more existing products, as we explore in our work. Further, our approach of verifying SPL improvement addresses this requirement by allowing us to evaluate the impact of SPL refactorings based on our theory for reasoning on *feature models*. Nevertheless, we believe that their theory for relating features and implementation modules may be complementary to ours for more ambitious applications of FM refactorings: given a systematic way of mapping FM constructs to software components, we can infer SPL refactorings on programs from analogous refactorings on corresponding feature models. Therefore, a model-driven approach to SPL refactoring would thus become feasible.

Another work [6] explores the application of refactoring to SPL Architectures. They present metrics for diagnosing structural problems in a SPL Architecture, and introduce a set of architectural refactorings that can be used to resolve those problems. These metrics can be useful for detecting bad smells. In contrast to our work, they apply the traditional notion of refactoring. Also, they do not propose a set of refactorings for FMs as in our work. A similar work [12] shows a case study in refactoring legacy applications into

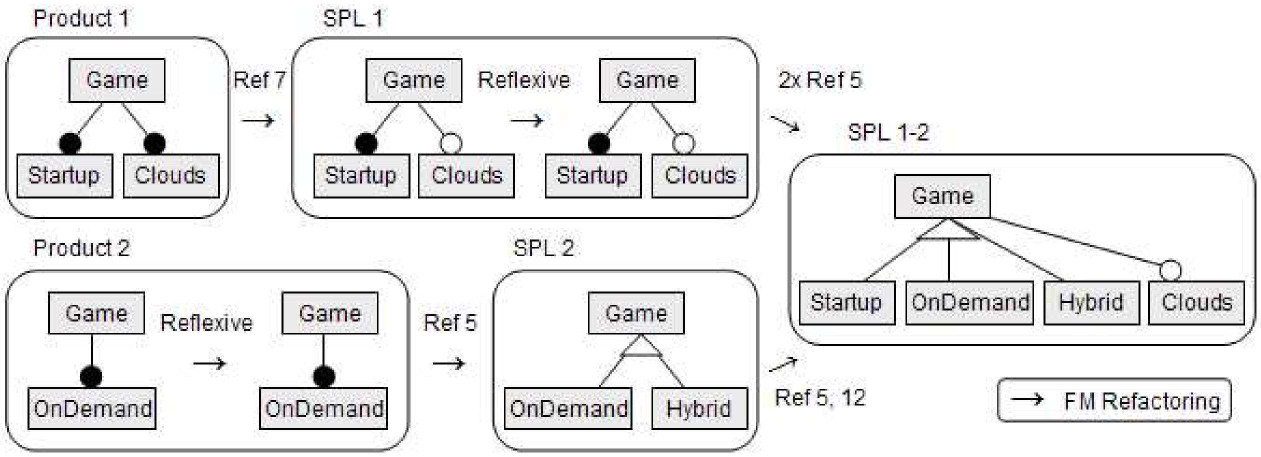


Figure 7. Case Study Feature Model Refactorings

SPL implementations. They defined a systematic process for refactoring products, in order to define SPLs. However, configurability of the resulting SPLs are only checked by testing; we believe that our approach for verifying configurability with FM refactorings can be useful in this process, especially when extracting a SPL from more than one product or SPL, improving reliability in the process. Also, they do not deal with refactoring in reactive contexts.

Batory [2] presents a semantics for FMs, connecting it to grammars and propositional formulas. The connection between FMs and propositional formulas enables the use of SAT solvers to perform a finite number of analysis. We specified and encoded in PVS a similar semantics for FM considering the same propositional formulas. We additionally proved, using the PVS prover, a number of refactorings. As mentioned by such author, FMs do not have unique representations as feature diagrams. By using our bidirectional refactorings, we show a reduction strategy (completeness result) that relates any two FM semantically equivalent. He is concerned with building a tool for FMs for checking specific properties; In our case, we can specify and prove not only specific but any kind of general property that holds for FMs.

Extensions to cardinality-based FMs can be found elsewhere [8], including a formal semantics to FMs with these features, translating FMs into context-free grammars. Our semantics could also be extended similarly, and new FM refactorings can be proposed for dealing with such cardinality. We also note that their formal treatment of FM specialization could be seen as the opposite of our notion of FM refactoring, except for the fact that our refactoring notion also relates multiple FMs.

Another work [18] proposes a textual language for describing features. Their language is similar to ours, but do not consider propositional formulas. They propose a notion of FM semantics that is equivalent to ours. Also, a set of fifteen rules relating equivalent FMs are proposed, which are very similar to our bidirectional refactorings. All proposed rules can be derived using our bidirectional refactorings following the reduction strategy (completeness result). These rules are not proven to be complete, as in our work. Additionally, they informally argue soundness, in contrast with our approach, which uses a theorem prover to increase the confidence. Moreover, we show that our set of bidirectional refactorings is minimal. So, one B-Refactoring cannot be derived using another B-Refactoring. If they had proposed a more general rule, similar to a refactoring for introducing or removing formulas, their Rules 1-4, for instance, would be derived from this more general rule. So, our set of B-Refactorings is more concise (minimal) and contain less transformations. Moreover, they do not use their FM refactorings to apply refactorings in SPL, as in our work. Their work also presents an option for configuration knowledge, mapping features to classes; our approach for verifying configurability improvement in Section 5 can be used in the presence of such option.

Another work [3] extends FMs in order to include constraints. They can automatically analyze five properties in this language, such as number of instances of a FM. However, they do not propose a set of refactorings for FM and use them to refactor SPL.

Refactoring *Merge Programs* introduced in Section 2.2 addresses building a product line from existing design/implementation artifacts. However, if high-level algebraic specification of products were available, as described in [15], an efficient optimization algorithm could be applied in order to extract the product line core from these specifications with the Shared Class Extractor operator [15]. However, the hypothesis of having this high-level specification may not be met in practice, in such a way that the domain engineer would need to address handling legacy software directly at the design or at the implementation level.

7. Conclusions

In this paper, we extended the traditional notion of refactoring to the SPL context. Besides traditional program refactoring, feature models are refactored. Not only the quality of the program is improved, but also the quality of the feature model, by maintaining or increasing its configurability. We showed a set of sound feature model refactorings and evaluated them in a real case study in the mobile domain.

As a future work, we aim at applying our approach in more case studies, in order to assess the benefits and limitations of this work in additional real projects. Moreover, from these case studies we can propose more FM refactorings, and make considerations on their usefulness. Finally, we intend to explore the role of configuration knowledge for achieving model-driven SPL refactoring.

Acknowledgments

We gratefully acknowledge Don Batory for working closely with us to improve this paper. We also thank the anonymous referees for useful suggestions. This research was partially sponsored by CNPq (grants 481575/2004-9, 141247/2003-7, 552068/2002-0), MCT/FINEP/CT-INFO (grant 01/2005 0105089400), and FAPERJ (grant E-26/151.493/2005).

References

[1] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Conference of Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer, 2005.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference of Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE)*, 3520:491–503, 2005.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unifi ed Modeling Language User Guide*. Addison-Wesley, 1999.

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

[6] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek. Refactoring product line architectures. In *IWR: Achievements, Challenges, and Effects*, pages 23–26, 2003.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

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

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] R. Gheyi, V. Alves, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Bidirectional refactorings: Catalog and completeness. Technical Report TR-UFPE-CIN-200608028, Federal University of Pernambuco, 2006.

[11] R. Gheyi, V. Alves, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Theory and proofs for feature model refactorings. Technical Report TR-UFPE-CIN-200608027, Federal University of Pernambuco, 2006.

[12] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 369–378. IEEE Computer Society, 2005.

[13] C. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th Workshop on Software Product-Family Engineering*, pages 282–293, 2001.

[14] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering*, pages 112–121. ACM Press, 2006.

[15] J. Liu and D. S. Batory. Automatic remodularization and optimized synthesis of product-families. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2004.

[16] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

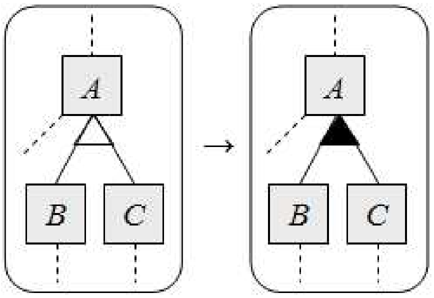
[17] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Germany, 1998. Springer-Verlag.

[18] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

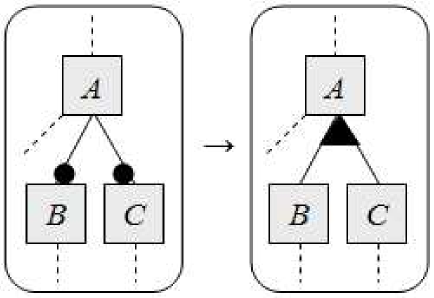
A. Unidirectional Refactorings Catalog

In this section, we present all unidirectional refactorings proposed. The f and $forms$ variables used in Refactoring 11 denote a formula and a set of formulas, respectively.

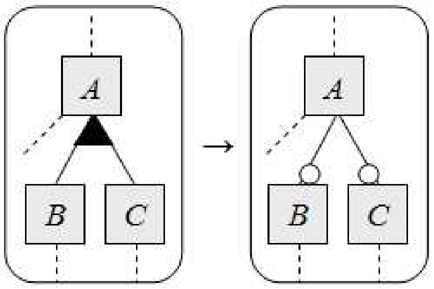
Refactoring 1. ⟨*convert alternative to or*⟩



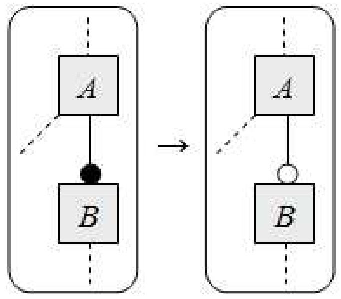
Refactoring 4. ⟨*add or between mandatory*⟩



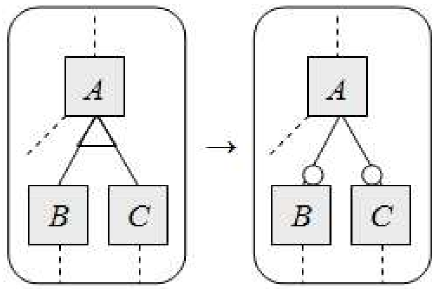
Refactoring 6. ⟨*convert or to optional*⟩



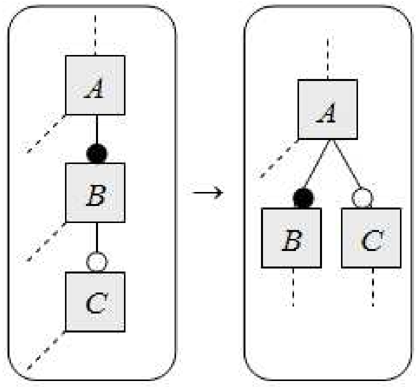
Refactoring 7. ⟨*convert mandatory to optional*⟩



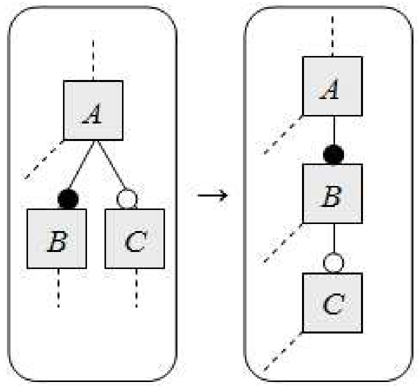
Refactoring 8. ⟨*convert alternative to optional*⟩



Refactoring 9. ⟨*pull up node*⟩



Refactoring 10. ⟨*push down node*⟩



Refactoring 11. ⟨*remove formula*⟩

$$\boxed{forms \wedge f} \rightarrow \boxed{forms}$$

Refactoring 12. ⟨*add optional node*⟩