Towards correct-by-construction product variants of a software product line: GFML, a formal language for feature modules

Thi-Kim-Zung Pham, Catherine Dubois, Nicole Lévy

Towards correct-by-construction product variants of a software product line: GFML, a formal language for feature modules

Thi-Kim-Zung Pham
Cedric, CNAM
Paris, France
pham_t43@auditeur.cnam.fr

Catherine Dubois
Cedric, ENSIIE
Évry, France
catherine.dubois@ensiie.fr

Nicole Levy
Cedric, CNAM
Paris, France
nicole.levy@cnam.fr

Software Product Line Engineering (SPLE) is a software engineering paradigm that focuses on reuse and variability. Although feature-oriented programming (FOP) can implement software product line efficiently, we still need a method to generate and prove correctness of all product variants more efficiently and automatically. In this context, we propose to manipulate feature modules which contain three kinds of artifacts: specification, code and correctness proof. We depict a methodology and a platform that help the user to automatically produce correct-by-construction product variants from the related feature modules. As a first step of this project, we begin by proposing a language, GFML, allowing the developer to write such feature modules. This language is designed so that the artifacts can be easily reused and composed. GFML files contain the different artifacts mentioned above. The idea is to compile them into FoCaLiZe, a language for specification, implementation and formal proof with some object-oriented flavor. In this paper, we define and illustrate this language. We also introduce a way to compose the feature modules on some examples.

1 Introduction

Software Product Line Engineering (SPLE) is a paradigm used to develop software-intensive systems that share common assets [14, 1]. In SPLE, a feature is used to represent a characteristic behavior as a unit of functionality of the software product line (SPL) [1]. Given a set of features, the configuration of a SPL is constructed by composing the features. Following generative programming mechanisms [4], the respective product variant can be derived automatically from a feature selection and artifacts of each selected feature. The product generation may be realized through Feature-Oriented Programming (FOP) techniques [7], in which each feature is mapped to a feature module containing its artifacts. The product variant is synthesized from the artifacts of the involved feature modules. In this context, we demonstrate our approach that helps the developer to write feature modules containing their artifacts using a dedicated and generic language.

Some authors proposed approaches for constructing product variants of a product line by reusing and synthesizing artifacts. Using design by contract [13] and adhering to FOP techniques, Thüm proposed a proof composition approach and strategies to reduce the effort of verification by reusing partial proofs [18] [17] [16]. The FEATUREHOUSE framework, described in [2], enables the construction of a new program from existing programs using the structure of the SPL feature tree. Although the above methods can implement software product line efficiently and mostly automatically, we still need a method to generate and prove correctness of all product variants more efficiently and automatically.

In this direction some advances have been proposed in the context of programming language meta-theory within the Coq proof assistant [10] [11]. However these tools are dedicated to a very specific

J. Atlee and S. Gnesi (Eds.): FMSPLE 2015
EPTCS 182, 2015, pp. 44–55, doi:10.4204/EPTCS.182.4

domain and still require an important expertise.

To tackle the above limitations, we depict a methodology that helps the user to automatically produce correct-by-construction product variants. This methodology aims at being independent of any concrete target language but at first we focus and experiment with FoCaLiZe, a language used to specify, implement and prove (`http://focalize.inria.fr`). In this paper, we propose a language, called **GFML** (for Generic Feature Module Language) allowing the developer to write feature modules which contain three kinds of artifacts: specifications, code and correctness proofs. GFML is inspired from FoCaLiZe but sticks to a FOP approach and tries to reduce the developer's effort. This language is designed so that the artifacts can be easily reused, composed and translated into different languages.

In Section 2, we first describe the background of our paper, in particular we present briefly FoCaLiZe. Then in Section 3, we illustrate our definition of feature module and a brief description of our language (GFML) used to write feature modules. In Section 4, we illustrate the translation of GFML into FoCaLiZe. In Section 5, we describe how to compose two feature modules by giving an example, and depict a composition framework for automatically generating correct-by-construction product variants. In Section 6, we review related work. Finally we conclude the paper and discuss future works in Section 7.

2 Background

2.1 Software Product Line

In SPLE, a **feature** is a characteristic behavior specified as a unit of functionality of a product line [1]. A set of features, called a **feature model**, is often graphically depicted as a tree, also called a **feature diagram**. In addition to the presentation of commonality and variability, a feature diagram also contains various relationships and additional constraints between features. The main relationships in feature diagrams are optional/mandatory, alternative/or and implies/excludes. Because of visual aid, feature diagrams are considered as standard representations that help the user to choose product configurations of a product line. In this work, we only focus on a simple form of feature diagrams (i.e. Figure 1) that can illustrate optional features.

Valid Configuration. A configuration of a product line is a set of features selected in the corresponding feature diagram. All the information such as relations and constraints inferred from the feature diagram is used to check the validity of the configuration. In our work, we only consider valid configurations. However, checking the validity of a configuration is out of the scope of this paper. Many checkers exist (e.g. [5], [8]) and we rely on them. As we start from valid configurations, we consider that the configurations contain all the features that are to be composed in order to build the final products. So the distinction between mandatory and optional features is not relevant here.

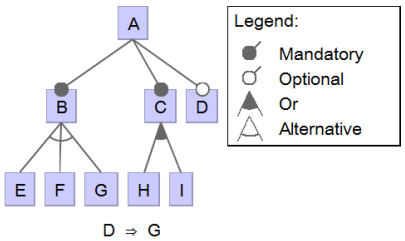


Figure 1: Feature diagram

2.2 Motivating Example

As a running example, we consider the bank account product line described in [17] whose feature diagram is shown in Figure 2. It illustrates a family of products allowing the management of bank accounts. The root feature *BankAccount* (BA for short) provides

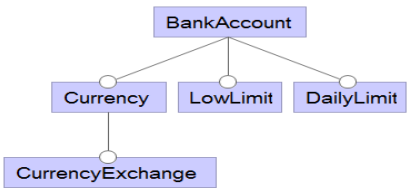


Figure 2: Feature diagram of bank account product line

the basic management of an account. It allows the bank storing the current balance and the amount of money added into or withdrawn from the account. A customer can withdraw more money from the account than available if it is within an over limit. The feature BA has three optional child features *DailyLimit* (DL for short), *LowLimit* (LL for short), *Currency*. The feature DL allows the bank to limit the amount of money withdrawn in a day while the other feature LL indicates that the bank only authorizes a customer to withdraw money from the account only if the amount is greater than a low limit. Related The feature *Currency* accommodates the management of currency. Finally, an optional feature *CurrencyExchange* is established as a child of *Currency* to enable the calculation of currency exchange.

2.3 Quick Presentation of FoCaLiZe

This subsection presents briefly the technical background necessary to understand the section about the translation of GFML to FoCaLiZe.

The FoCaLiZe (`http://focalize.inria.fr`) environment provides a set of tools to describe and implement functions and logical statements together with their proof. A FoCaLiZe source program is analyzed and translated into OCaml sources for execution and Coq sources for certification. The FoCaLiZe language has an object oriented flavor allowing inheritance, late binding and redefinition. These characteristics are very helpful to reuse specifications, implementations and proofs.

A FoCaLiZe specification can be seen as a set of algebraic properties describing relations between input and output of the functions implemented in a FoCaLiZe program. For writing code, FoCaLiZe offers a pure functional programming style close to ML, featuring strong typing, recursive functions, data types and pattern-matching. Proofs written using the FoCaLiZe proof language are sent to the Zenon prover which produces proofs that can be verified by Coq for more confidence [9]. The FoCaLiZe proof language is a declarative language in which the programmer states a property and gives hints to achieve its proof which is performed by Zenon.

FoCaLiZe units are called *collections*. They contain entities in a model akin to classes and objects or types and values. Collections have functions and properties which can be called using the "`!`" notation. They are derived from other units called *species* which specify and implement functions.

A species defines a set of entities together with functions and properties applying to them. At the beginning of a development, the representation of these entities is usually abstract, it is precised later in the development. However the type of these entities is referred as `Self` in any species. Species may contain specifications, functions and proofs. More precisely species may specify a function or a property (with resp. `signature`, `property` keywords) or implement them (`let` keyword when a function is defined, `proof of` keyword to introduce a proof of a property). A `let` defined function must match its signature and similarly a proof introduced by `proof of` should prove the statement given by the `property` keyword. Statements belong to first order typed logic.

As said previously, FoCaLiZe integrates inheritance, late binding and redefinition to ease reuse and modularity. Inheritance allows the definition of a new species from one or several other species. The new species inherits all the functions, properties and proofs of its parents. Some syntactical mechanisms are provided to prevent ambiguities. A species may provide a definition for a function that is only specified in its parents. It may also redefine a function when this one is already defined in a parent but in that case the signature is maintained (no overloading). Multiple inheritance comes with a late binding mechanism close to the one found in object oriented languages (even if the resolution is statically done by the compiler).

A collection is a species where every specified function is defined and every property is proved (or explicitly admitted). Furthermore, in a collection, the concrete *representation* of entities is made

private and a programmer using the collection can only use its functions and properties according to the interface. Consequently building a collection from a species provides encapsulation. Species may have parameters which may either be collections or entities providing in that way parametric polymorphism. This parametrization will be intensively used in the translation of GFML to FoCaLiZe (see Section 4).

For more details on FoCaLiZe please refer to the reference manual. A more thorough overview can be found in [3].

3 Artifacts

3.1 Feature Module

We adhere to the FOP technique that suggests to map each feature to a separate **feature module** that implements the feature. In our context, each feature module consists of its **artifacts**: specifications, code and correctness proofs. Specifications are given as a set of expected properties or requirements. Technically these properties are logical formulas relating together some functions described only by their signature. Thus in our setting, a specification is close to an algebraic data-type. Code has to be understood here as the implementation of the functions introduced/declared in the specifications. We place ourselves in a functional programming setting. The proofs here concern the correctness of the code with respect to the specifications.

- **Specification artifact** includes the function declarations and the properties. A function declaration or signature only describes the name of the function and the type of its arguments and result. A property is a (first order typed) logical formula. A new property is expressed by a new logical formula while a refining property refers to existing properties of the *parent* feature module and may add new premises and conclusions.
- **Code artifact** consists of the definition of the concrete representation type r and the function definition/redefinitions df. The representation type is the concrete type associated to the abstract data-type specified in the specification artifact. It will be a concrete type (basic or complex types, *à la ML*) or a Cartesian product of concrete types. The representation type is unique for each feature module and can be also constructed from the existing representation type of *parent*. A function is defined/redefined after the representation type has been defined.
- **Proof artifact** contains proofs pf. Each proof corresponds to a property. While writing the proof for a refining property, the mentioned properties of *parent* can be reused to support the proving process.

We define a feature module fm as a 5-tuple: function declarations d, properties p, representation type r, function definitions/redefinitions df and proofs pf.

$$fm = (d, p, r, df, pf)$$

3.2 Description of GFML

We propose a generic language, called **GFML**, to write these feature modules. Each feature module is embedded into a separate file *.gfm*. This language is suitable for all feature modules possibly containing three kinds of artifacts: specifications, code and correctness proofs.

This language is inspired from FoCaLiZe, in particular the styles for writing specifications, code and proofs are common. GFML and FoCaLiZe mainly differ in the way to structure and organize information.

```
1 f_module BA
2  signature over: int;
3  signature balance: BA -> int;
4  signature update: BA -> int -> BA * S;

5  property update_succ_BA: all x : BA, all a :
6 int, all r : BA * S =
7    (balance(x) + a) >= over -> update (x,a) = r
8    -> (balance(first(r)) = (balance(x) + a)) &&
9 (second(r) = success);

10  representation = int;

11  let balance(x) = x;
12  let update(x,a) =
13   if ((balance(x) + a) >= over) then
14    ((balance(x) + a), success)
15   else (x, no_success);

16  proof of update_succ_BA = (by definition of
17 update, balance);
```

(a) Module BA

```
1 f_module DL from BA
2  signature limit_withdraw: int;
3  signature withdraw: DL -> int;

4  property update_succ_DL: all x : DL, all a :
5 int, all r : DL * S =
6   refines BA!update_succ_BA(x, a, r)
7   new premise: H1
8   new conclusion: (withdraw(first(r)) =
9 withdraw(x) + a);
10  property update_no_succ_DL: all …

11  representation = int*BA!representation;

12  let withdraw(x) = second(x);
13  let update(x,a) = let w = withdraw(x) in
14   let rr = BA!update(x,a) in
15   let n_b = first(rr) in
16   let n_s = second(rr) in
17   let n_w = w + a in
18    if (a <= 0) then
19     if ((n_w >= limit_withdraw) && (n_s =
20 success)) then
21       ((n_b,n_w),n_s)
22     else (x,no_success)
23    else ((n_b,w),n_s);

24  proof of update_succ_DL = (by hypothesis H2
25 definition of withdraw, update, make balance
26 property BA!update_succ_BA);
27  proof of update_no_succ_DL = (by …);
```

(b) Module DL

Figure 3: Feature modules in GFML

Our main objective in defining GFML is to propose a language close to FoCaLiZe that already allows for the three kinds of artifacts in a single setting but closer to the description of commonality and variability we can find in feature diagrams. As we will see in Section 4, the expression of a feature module is much simpler that its translation in FoCaLiZe. GFML is also inspired from design by contract applied to FOP as in [15]. In the same way, GFML syntax allows the programmer to focus in a GFML feature module on the modifications brought to the specifications or implementations of the parent.

3.3 Examples

Corresponding to the feature diagram of the bank account product line given in Figure 2, the feature module BA implements the root feature BA. Feature modules DL, LL, and *Currency* are mapped from the features DL, LL, and *Currency* respectively. They have a common parent, i.e. the root module BA. Feature module *CurrencyExchange* corresponding to feature *CurrencyExchange*, is a child of the module *Currency*.

The root feature module BA, written with GFML, is shown in Figure 3a. This module includes three signatures: *over* - is over limit, *balance* - gets the current balance value of the account and *update* - upgrades the new value balance and also returns the status of the operation (*success* or *no_success*, of type *S* whose definition is omitted here). The next part of the module BA contains the property *update_succ_BA* (line 5) that specifies a customer can withdraw more money *a* from the account than available if the balance is within *over*. In that case, its status must be *success*. The primitive functions *first* and *second* used in this property are the usual projections of a Cartesian product. Then the representation type is defined as *int* (line 10), it means that an account is only represented by its balance. Then appear the definitions of the functions *balance* and *update* (lines 11-15). The proof of property *update_succ_BA*

includes two proof hints: by definition of *update* and *balance* definitions. This means that the proof must be done by unfolding the definitions of both functions.

Another example of GFML feature module is the module DL defined according to its *parent* feature module BA (Figure 3b). Two new declarations *limit_withdraw* and *withdraw* are added into the module (lines 2-3). The module introduces the constant *limit_withdraw* only declared at that point. It denotes the limit of withdrawn money in a day. It also introduces another function *withdraw* that returns, for an account, the current amount of withdrawn money in a day. The functions *update*, *balance* and *over* defined in the parent are also available in the present feature module. A new property *update_succ_DL* is obtained by modifying the property *update_succ_BA* from the *parent* (line 6). This modification includes a new premise called H1 (line 7) for short in the figure (but announced below) and a new conclusion (lines 8-9). The premise H1 is expressed as follows:

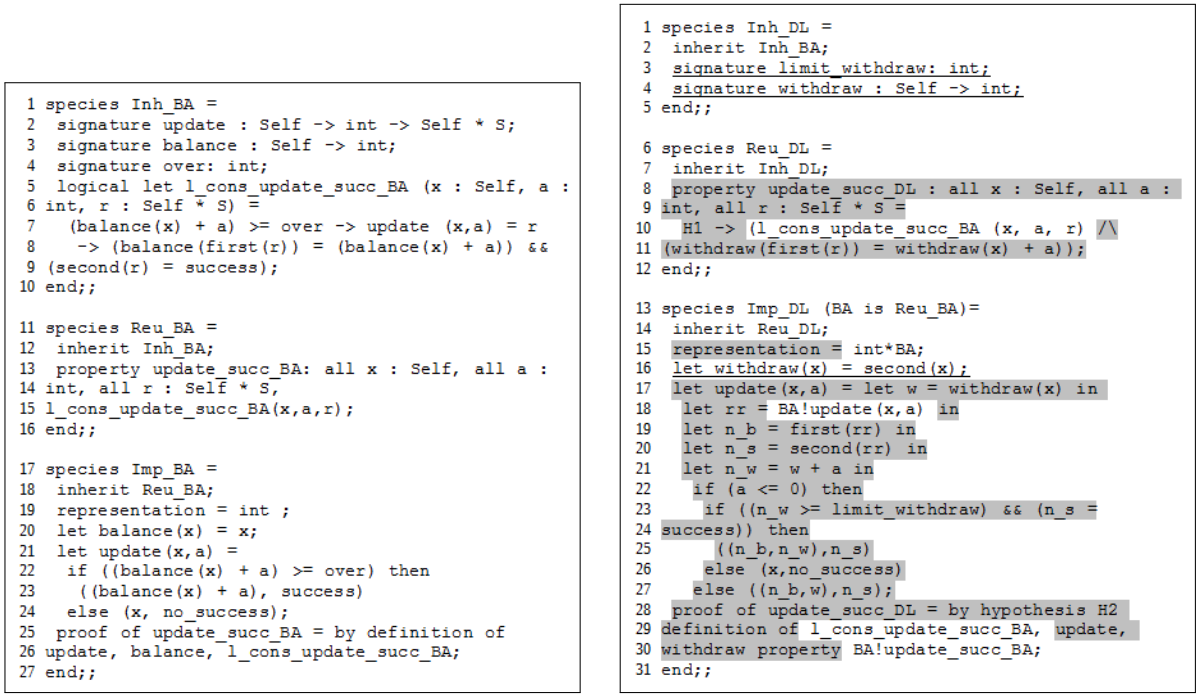
$$r = update(x,a) \rightarrow (a <= 0) \rightarrow (all\ \ n\_w\ \ w : int, all\ \ n\_s : S,\ \ withdraw(x) = w\ \ \&\&\ \ w + a = n\_w\ \ \&\&\ \ second(BA!update(x,a)) = n\_s \rightarrow (n\_w >= limit\_withdraw)\ \ \&\&\ \ (n\_s = success))$$

It states that the bank allows a customer to withdraw money only if the amount of withdrawn money in a day is greater than *limit_withdraw* (in this case *w*, *n_w* and *limit_withdraw* are negative numbers). The new conclusion states that this operation has to modify the account by updating the amount of withdrawn money. The representation type of module DL is defined as a Cartesian product of *int* (i.e. the concrete type associated with the amount of money withdrawn in a day) and the representation type of the *parent* (line 11). The functions *withdraw* and *update* are defined/redefined (lines 12-23). We can notice that in the redefinition of *update* the call to the parent function *update* is done with the parameter *x*, there is here an implicit conversion that will be inserted during the translation into FoCaLiZe. The proof of the property *update_succ_DL* reuses the property *update_succ_BA* as a proof hint (line 26). All modifications and additions compared to the parent module BA are highlighted or underlined.

4 From GFML to FoCaLiZe

In this section, we illustrate the translation from GFML to FoCaLiZe. A GFML feature module is mapped into three FoCaLiZe separate *species*. The first *species* ***Inh***_ contains all desired elements that are inherited in the child module. These elements can be function declarations, properties or predicates. A predicate, which is introduced with the keyword '*logical let*' and named *l_cons*, is associated with each property *p* found in a module. For example, the feature module BA written in Figure 4a is mapped to the species *Inh_BA* (line 1). A predicate *l_cons_update_succ_BA* (line 5) is associated with the property *update_succ_BA*. In Figure 4b, the species *Inh_DL* of the child DL inherits the species *Inh_BA* of module BA (line 2).

The FoCaLiZe inheritance mechanism allows the programmer to reuse the artifacts without any change (except for functions that can be redefined without changing its signature). But although some properties may be kept in the child module, others may be modified by adding new premises or/and new conclusions. In this last case, the FoCaLiZe inheritance mechanism is not sufficient. We have to use the parametrization facility to encode the right meaning. To tackle this limitation, we propose the second *species* ***Reu***_ that inherits the first *species* ***Inh***_. It includes all desired specifications that are reused to describe new ones in the child. Each property specified in this *species* must have a corresponding predicate *l_cons* in the first *species* ***Inh***_. Instead of mentioning the property, we use this predicate for expressing new properties in the child. This trick is used because FoCaLiZe does not allow the modification of an inherited property in a *species*. For example, in Figure 4a *species Reu_BA* inherits



```
1 species Inh_BA =
2  signature update : Self -> int -> Self * S;
3  signature balance : Self -> int;
4  signature over: int;
5  logical let l_cons_update_succ_BA (x : Self, a :
6 int, r : Self * S) =
7    (balance(x) + a) >= over -> update (x,a) = r
8     -> (balance(first(r)) = (balance(x) + a)) &&
9 (second(r) = success);
10 end;;

11 species Reu_BA =
12  inherit Inh_BA;
13  property update_succ_BA: all x : Self, all a :
14 int, all r : Self * S,
15 l_cons_update_succ_BA(x,a,r);
16 end;;

17 species Imp_BA =
18  inherit Reu_BA;
19  representation = int ;
20  let balance(x) = x;
21  let update(x,a) =
22   if ((balance(x) + a) >= over) then
23    ((balance(x) + a), success)
24   else (x, no_success);
25  proof of update_succ_BA = by definition of
26 update, balance, l_cons_update_succ_BA;
27 end;;
```

(a) Module BA in FoCaLize

```
1 species Inh_DL =
2  inherit Inh_BA;
3  signature limit_withdraw: int;
4  signature withdraw : Self -> int;
5 end;;

6 species Reu_DL =
7  inherit Inh_DL;
8  property update_succ_DL : all x : Self, all a :
9 int, all r : Self * S =
10   H1 -> (l_cons_update_succ_BA (x, a, r) /\
11 (withdraw(first(r)) = withdraw(x) + a));
12 end;;

13 species Imp_DL (BA is Reu_BA)=
14  inherit Reu_DL;
15  representation = int*BA;
16  let withdraw(x) = second(x);
17  let update(x,a) = let w = withdraw(x) in
18   let rr = BA!update(x,a) in
19   let n_b = first(rr) in
20   let n_s = second(rr) in
21   let n_w = w + a in
22    if (a <= 0) then
23     if ((n_w >= limit_withdraw) && (n_s =
24 success)) then
25      ((n_b,n_w),n_s)
26     else (x,no_success)
27    else ((n_b,w),n_s);
28  proof of update_succ_DL = by hypothesis H2
29 definition of l_cons_update_succ_BA, update,
30 withdraw property BA!update_succ_BA;
31 end;;
```

(b) Module DL in FoCaLize

Figure 4: Feature modules in FoCaLize

the *species Inh_BA*. It contains the property *update_succ_BA* (line 13) that is related to the predicate *l_cons_update_succ_BA* in *species Inh_BA*. This predicate is reused to specify the refining property *update_succ_DL* of module DL (line 10 of Figure 4b).

The parametrization mechanism in FoCaLize is used here to circumvent the fact that in FoCaLiZe when the representation type is fixed in a species P, it cannot be changed in any species which inherits P. So in order to build a new species that can reuse functions, properties, predicates and proofs that are defined for BA, we have to parametrize this species by an implementation of BA. Thus, when the DL feature is selected, the account is implemented as a pair containing a basic account of type BA and an integer corresponding to the new attribute, i.e. the amount of money withdrawn in a day. The implementation of the functions *balance* and *over* are, in this context, just a call to the parent's functions combined with the first projection. For example, *species Imp_BA* inherits *species Reu_BA* (line 18 of Figure 4a) while *species Imp_DL* inherits *species Reu_DL* (line 14 of Figure 4b). A parameter *BA* encapsulates *species Reu_BA* (line 13 of Figure 4b). It is used to call function *update* of module BA (line 18). In the proof of the refining property *update_succ_DL*, property *update_succ_BA* of the parent BA is considered as a proof hint (line 30). Its associated formula *l_cons_update_succ_BA* is also a proof hint (line 29). Let us notice that Zenon, the FoCaLiZe prover has done all the proofs with the given hints.

We can notice that the use of a language such as FoCaLiZe for implementing and verifying feature modules is quite complex. The proposed language GFML is a solution to write the feature modules together with their artifacts more easily. We have followed with success this translation scheme for all the features appearing on the feature diagram of Figure 2. For the moment, the translation is done manually.

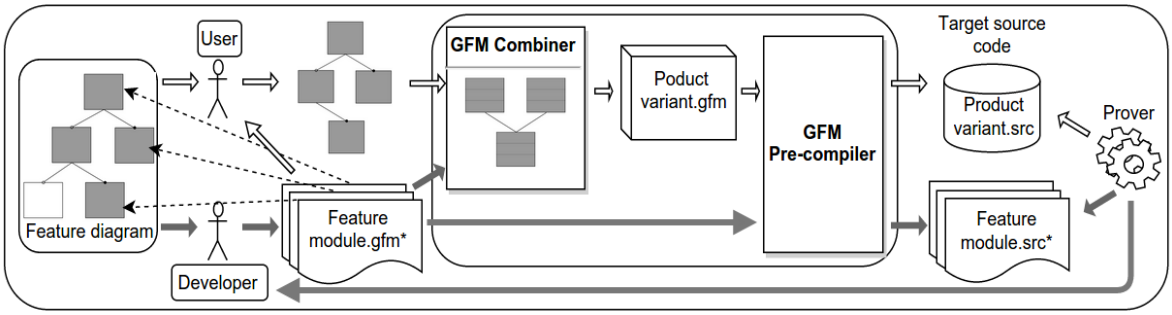


Figure 5: Methodology

5 Towards Feature Module Composition

5.1 Methodology for composing artifacts

The work previously described is the first step in the proposition of a methodology to help the user to automatically produce correct-by-construction product variants from features selected in a feature diagram. This methodology is illustrated in Figure 5. We assume a SPL is described with a feature diagram. First, the developer writes the different feature modules with the GFML language. He uses the **GFM Pre-compiler** to translate his/her feature modules into FoCaLiZe to verify them (using the Zenon prover). Once this work has been done, the user - he may be different from the developer - chooses features from the feature diagram. Based on this selection, the corresponding configuration is determined. The related GFML feature modules are then sent to the **GFM Combiner** that will compose the feature modules of the configuration in order to obtain a composition feature module that will be translated into the desired product variant, which by composition is correct by construction. We expect this method to be independent from the target language which is required to be able to express specifications and code and also proofs (unless these ones can be automatically produced).

In the next subsection we explain the main ideas of our composition mechanism on the running example.

5.2 Composition of two feature modules: an example

Some problems may appear when composing two feature modules, in particular conflicts may appear when synthesizing their artifacts. For example, which properties should be performed first? Which synthesized representation type will make less complex writing? To solve these problems, we suggest the user gives a composition order of the modules. In our work, we only consider the modified elements such as refining properties, representation types, function redefinitions or proofs of refining properties.

Similar to module DL, module LL written in Figure 6a is extended from the root module BA. A new signature *limit_low* declares the low limit of the account. A new property *update_no_succ_LL* is specified in line 7 and its proof is indicated in line 16. However, we only consider the modifications that can cause conflicts when composing two modules. For example, let us compose module LL with module DL. Their composition is the module LL_DL given in Figure 6b. Property *update_succ_LL* of module LL includes two parts (lines 3-6 of Figure 6a). Its highlighted part includes a new premise (shortened by H2), expressed as follows:

$$r = update(x,a) \rightarrow ((a >= 0) \;||\; (a <= limit\_low))$$

```
1 f_module LL from BA
2  signature limit_low: int;

3  property update_succ_LL: all x : LL, all a :
4 int, all r : LL * S =
5    refines BA!update_succ_BA(x, a, r)
6    new premis: H2;
7  property update_no_succ_LL: all …

8  representation = BA!representation;

9  let update(x,a) =
10  if ((a >= 0) || (a <= limit_low)) then
11    BA!update(x,a)
12  else (x, no_success);

13  proof of update_succ_LL = (by hypothesis H1
14 definition of update, balance property BA!
15 update_succ_BA);
16  proof of update_no_succ_LL = (by …);
```

(a) Module LL

```
1 f_module LL_DL from LL, DL
2  property update_succ_LL_DL : all x : LL_DL, all
3 a : int, all r : LL_DL * S =
4    refines DL!update_succ_DL(x, a , r )
5    new premise: r = update(x,a) ->
6     ((a >= 0) || (a <= limit_low));

7  representation = DL!representation;

8  let update(x, a) =
9   if (a >= 0) || (a <= limit_low) then
10    DL!update(x, a)
11  else (x, no_success);

12  proof of update_succ_LL_DL = (by hypothesis H1
13 defintion of update, balance property DL!
14 update_succ_DL, BA!update_succ_BA);
```

(b) Module LL_DL

Figure 6: Composition of feature modules

The premise specifies that a customer can withdraw money *a* from the account only if *a* is less than *limit_low* (in this case *a* and *limit_low* are negative numbers). The property *update_succ_BA* from the *parent* BA is its remaining part. Following the composition order, the composite property of two properties *update_succ_LL* and *update_succ_DL* is synthesized by taking H2 first and then mixing with *update_succ_DL*. It is named *update_succ_LL_DL* and embedded into a composition feature module LL_DL (lines 2-6 of Figure 6b). Similarly, the composite representation type is synthesized in line 7. The function *update* is redefined (lines 8-11). The proof of the composite property *update_succ_LL_DL* reuses two properties *update_succ_DL* and *update_succ_BA* as its proof hints (lines 13-14) indicate. The highlighted parts of module LL (Figure 6a) are the parts highlighted in module LL_DL (Figure 6b). The other parts which are not highlighted in Figure 6b are taken from module DL.

6 Related Work

Recently, many authors focused on constructing and verifying program variants in SPL. Together with these, composition methods of feature artifacts are offered. In this subsection, we compare to some of them.

Thüm et al., in [18], presents a composition approach where partial proofs are given in features and then composed to build the complete proofs for an individual product. The specifications are expressed using design by contract [13]. The Krakatoa/Why tool [12] is used to generate proof obligations that are exported to the Coq proof assistant where proofs are done and verified.

Our work is in line with the previous approach but it is dedicated to a functional setting whereas Thüm et al's work involves object-oriented programs. Specification artifacts - expressed as logical formulas - found in a GFML feature module are close to contracts. For example, a refining property is close to a refining contract since it includes a former specification and may add a new premise and a new conclusion. However in our work, each proof is complete. It is extended in a new proof in the result of a composition and thus in the resulting product variant.

Another composition framework FEATUREHOUSE is offered in [2]. Using the FST (feature structure tree) model, existing artifacts can be composed to construct a new program. The artifacts must have tree structures. In contrast, we presented a framework that allows us to construct products from

the artifacts of the features selected by the user. We are interested in expressing feature compositions as algebraic expressions using composition operations for each kind of artifacts instead of relying on the tree structure. Similar ideas were mentioned in [7] [6]. However, these approaches only focus on constructing products but do not mention how to verify them.

Recent researches [10] [11] had proposed some advances in feature composition in the context of meta-theory. However, these tools are dedicated to a very specific domain. Similar to Thum's work, the products in these approaches are verified in Coq. In contrast, implementing feature modules independently of any concrete target language is the purpose of our work. By proposing a generic formal language (GFML) for feature module, the artifacts are easy to reuse and synthesize but don't belong to any concrete language.

7 Conclusion

In this paper we have described a first step towards the production of a methodology allowing for the development of correct-by-construction product variants according to a FOP paradigm. The contribution is here the definition of the language GFML allowing the developer to write feature modules containing their specifications, code and correctness proofs.

These modules are translated to FoCaLiZe for verification and also for obtaining OCaml operational code. Consequently the product variants we are aiming at will be implemented in OCaml. GFML is here introduced to help the developer when he is writing the different feature modules, allowing him to describe the artifacts of a feature with respect to the artifacts of its parents. Realizing this directly in FoCaLiZe is a difficult task as it is exemplified by the translation scheme presented in a previous section. For the moment, nothing is implemented, all the examples found in this paper (all the possible configurations of the case study) have been obtained by a manual translation and composition.

Next step is to provide an implementation for the GFML Pre-compiler that will automatically translate GFML to FoCaLiZe. Then composition of feature modules will be formally defined and implemented in the GFML Combiner.

Another important perspective is to make our methodology independent of the target languages. Regarding this point, an intermediate step could be to adapt our methodology and tools to B or EventB where inheritance is not available.

References

[1] Sven Apel, Don S. Batory, Christian Kästner & Gunter Saake (2013): *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, doi:10.1007/978-3-642-37521-7.

[2] Sven Apel, Christian Kästner & Christian Lengauer (2009): *FEATUREHOUSE: Language-independent, automated software composition*. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, IEEE, pp. 221–231, doi:10.1109/ICSE.2009.5070523.

[3] Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, Lionel Habib, Thérse Hardin, Jaume Mathieu, Charles Morisset, François Pessaux, Renaud Rioboo & Pierre Weis (2008): *Trusted Software within Focal*. In: *C&ESAR 2008, Computer Electronics Security Applications Rendezvous*, pp. 162–179.

[4] Barbara Barth, Gregory Butler, Krzysztof Czarnecki & Ulrich W. Eisenecker (2001): *Generative Programming*. In Ákos Frohner, editor: *Object-Oriented Technology ECOOP 2001 Workshop Reader, ECOOP 2001 Workshops, Panel, and Posters, Budapest, Hungary, June 18-22, 2001, Proceedings*, *Lecture Notes in Computer Science* 2323, Springer, pp. 135–149, doi:10.1007/3-540-47853-1_11.

[5] Don S. Batory (2005): *Feature Models, Grammars, and Propositional Formulas*. In J. Henk Obbink & Klaus Pohl, editors: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, *Lecture Notes in Computer Science* 3714, Springer, pp. 7–20, doi:10.1007/11554844_3.

[6] Don S. Batory, Peter Höfner & Jongwook Kim (2011): *Feature interactions, products, and composition*. In Ewen Denney & Ulrik Pagh Schultz, editors: *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, ACM, pp. 13–22, doi:10.1145/2047862.2047867.

[7] Don S. Batory, Jacob Neal Sarvela & Axel Rauschmayer (2003): *Scaling Step-Wise Refinement*. In Lori A. Clarke, Laurie Dillon & Walter F. Tichy, editors: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, IEEE Computer Society, pp. 187–197, doi:10.1109/ICSE.2003.1201199.

[8] David Benavides, Pablo Trinidad & Antonio Ruiz Cortés (2013): *Automated Reasoning on Feature Models*. In: *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, Springer, pp. 361–373, doi:10.1007/978-3-642-36926-1_29.

[9] Richard Bonichon, David Delahaye & Damien Doligez (2007): *Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs*. In Nachum Dershowitz & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, *Lecture Notes in Computer Science* 4790, Springer, pp. 151–165, doi:10.1007/978-3-540-75560-9_13.

[10] Benjamin Delaware, William R. Cook & Don S. Batory (20110.1007/978-3-642-37521-71): *Product lines of theorems*. In Cristina Videira Lopes & Kathleen Fisher, editors: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, ACM, pp. 595–608, doi:10.1145/2048066.2048113.

[11] Benjamin Delaware, Steven Keuchel, Tom Schrijvers & Bruno C. d S. Oliveira (2013): *Modular monadic meta-theory*. In Greg Morrisett & Tarmo Uustalu, editors: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, ACM, pp. 319–330, doi:10.1145/2500365.2500587.

[12] Claude Marché, Christine Paulin-Mohring & Xavier Urbain (2004): *The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML*. *J. Log. Algebr. Program.* 58(1-2), pp. 89–106, doi:10.1016/j.jlap.2003.07.006.

[13] Bertrand Meyer (1992): *Ap10.1007/978-3-642-37521-7plying "Design by Contract"*. *IEEE Computer* 25(10), pp. 40–51, doi:10.1109/2.161279.

[14] Klaus Pohl, Günter Böckle & Frank van der Linden (2005): *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, doi:10.1007/3-540-28901-1.

[15] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel & Gunter Saake (2012): *Applying Design by Contract to Feature-Oriented Programming*. In: *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, *Lecture Notes in Computer Science* 7212, Springer, pp. 255–269, doi:10.1007/978-3-642-28872-2_18.

[16] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer & Gunter Saake (2014): *A Classification and Survey of Analysis Strategies for Software Product Lines*. *ACM Comput. Surv.* 47(1), p. 6, doi:10.1145/2580950.

[17] Thomas Thüm, Ina Schaefer, Martin Hentschel & Sven Apel (2012): *Family-based deductive verification of software product lines*. In Klaus Ostermann & Walter Binder, editors: *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, ACM, pp. 11–20, doi:10.1145/2371401.2371404.

[18] Thomas Thüm, Ina Schaefer, Martin Kuhlemann & Sven Apel (2011): *Proof Composition for Deductive Verification of Software Product Lines*. In: *Fourth International IEEE Conference on Software Testing,*

Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, IEEE Computer Society, pp. 270–277, doi:10.1109/ICSTW.2011.48.