# Correctness-by-Construction for Feature-Oriented Software Product Lines

Tabea Bordis
t.bordis@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Tobias Runge
tobias.runge@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Ina Schaefer
i.schaefer@tu-bs.de
TU Braunschweig
Braunschweig, Germany

## Abstract

*Software product lines* are increasingly used to handle the growing demand of custom-tailored software variants. They provide systematic reuse of software paired with variability mechanisms in the code to implement whole product families rather than single software products. A common domain of application for product lines are safety-critical systems, which require *behavioral correctness* to avoid dangerous situations in-field. While most approaches concentrate on post-hoc verification for product lines, we argue that a stepwise approach to create correct programs may be beneficial for developers to manage the growing variability. *Correctness-by-construction* is such a stepwise approach to create programs using a set of small, tractable refinement rules that guarantee the correctness of the program with regard to its specification. In this paper, we propose the first approach to develop correct-by-construction software product lines using feature-oriented programming. First, we extend correctness-by-construction by two refinement rules for variation points in the code. Second, we give a proof for the soundness of the proposed rules. Third, we implement our technique in a tool called VarCorC and show the applicability of the tool by conducting two case studies.

***CCS Concepts:*** • **Software and its engineering** → **Formal software verification**; *Software product lines*; • **Theory of computation** → Hoare logic.

***Keywords:*** correctness-by-construction, software product lines, feature-oriented programming, formal verification

**ACM Reference Format:**
Tabea Bordis, Tobias Runge, and Ina Schaefer. 2020. Correctness-by-Construction for Feature-Oriented Software Product Lines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426959

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

*GPCE '20, November 16–17, 2020, Virtual, USA*

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8174-1/20/11...$15.00
https://doi.org/10.1145/3425898.3426959

## 1 Introduction

*Software product line engineering* [32] is increasingly used in both academia and industry. Consequently, verification techniques for the resulting *software product lines* have been a big topic in research for the last decade [13, 25, 36]. Software product lines are families of related programs that share a common code base and are each composed by a valid configuration of artifacts [17]. Thereby, the common and varying parts of a product line are defined by features. Advanced techniques often use mechanisms to reuse code and variable code structures to generate the different variants [5, 20, 35]. Also in the domain of safety-critical systems, products lines are used more often [27]. As a result, behavioral correctness of these product lines is an important concern.

While most approaches in research apply post-hoc verification [11, 23, 39], where a program is verified after its implementation, we claim that a stepwise approach to create correct programs can be beneficial. This benefit arises especially when specifying highly variable software, as the variable implementation is created incrementally together with its specification. In contrast, with post-hoc verification the developer has to specify and verify all variable parts *after* finishing the whole implementation which becomes even more difficult with growing variability. Therefore, we focus on *correctness-by-construction* as pioneered by Dijkstra, Gries, or Kourie and Watson [19, 22, 26] as a stepwise approach to create correct programs. At first, the specification in form of pre- and postcondition is defined which is then refined into code using a set of tractable *refinement rules*.

In previous work, *variational correctness-by-construction* has been introduced as an extension of correctness-by-construction to enable the development of single variational methods [10]. Therefore, a refinement rule that uses a variability mechanism similar to the software product line mechanism called *feature-oriented programming* [5] has been proposed. However, the developer needed to define the refinements for every method manually, whereas in product lines there is usually a feature model which globally defines the relationships of features and thus for all methods automatically.

The global definition of relationships of features also enables the interaction between multiple variational methods in form of variational method calls in the product line, which neither is supported by variational correctness-by-construction.

In this paper, we build on variational correctness-by-construction to raise the scope from developing single variational methods to the development of whole software product lines with correctness-by-construction. Thereby, we propose the first approach on correctness-by-construction for feature-oriented software product lines that will serve as baseline for future extensions and improvements.

After giving background knowledge about software product lines and correctness-by-construction in Section 2 and introducing a motivating example that is used throughout this paper in Section 3, we present our contribution in Section 4. First, in Section 4.1, we give an introduction into contract composition as mechanism to derive variable contracts for the variants in a product line. Second, in Section 4.2 and 4.3, we extend correctness-by-construction by two refinement rules that cover the variability in the code of the product line and explicitly define the set of valid refinements as given by the feature model. This enables the reasoning of correctness for the whole product line including interactions between multiple variational methods using method calls. Third, we give a proof of soundness for the proposed refinement rules in Section 4.4. To evaluate our approach in Section 5, we implemented it in a tool called VarCorC[1] and used it to create two software product lines as case studies showing feasibility. In summary, we make the following contributions:

- We extend correctness-by-construction by two refinement rules for original calls and variational method calls.
- We prove both refinement rules correct by using structural induction.
- We provide a prototypical implementation in an open-source tool called VarCorC.
- We show feasibility on two case studies.
- We share two case studies that have been created using VarCorC as benchmark for further research.

## 2 Background

In this section, we give necessary background for our work. Therefore, in Section 2.1, we introduce software product lines and give formal definitions for the specific terms as a basis to our contribution. In Section 2.2, we present correctness-by-construction, which is the approach that we extend to create correct software product lines.

### 2.1 Software Product Lines

A *software product line* is a set of software products as variations of a common code base [14]. It systematically reuses software artifacts using variable code structures, so that they can be assembled according to the needs of the user. Consequently, it allows to implement a whole product family rather than single software products. There are two processes in software product line engineering, called *domain engineering* and *application engineering* [32]. In domain engineering, the commonalities and differences of the different products in the family are defined and realized. They are communicated as *features*. The definition of a feature often varies in the literature depending on the viewpoint [12]. We simply define a feature as a label. In the application engineering process, the feature configuration is created adhering to the restrictions of the feature model. Afterwards, the selected features are composed with the corresponding technique realizing the features in the domain engineering process to form the finished variant of the product line.

The features of a product line are usually organized in *feature models* [24]. They can be defined as mandatory, variable or collected in groups, which leads to complex dependencies that are often captured in tree structures. For simplicity, we refer to a feature model as the set of all valid combinations of features that result from the modeled dependencies. One valid combination of features is called *feature configuration*.

**Definition 2.1** (Feature Model)**.** Let $\mathbb{F}$ be the universe of feature labels. A feature model FM is a set of valid configurations $FM \subseteq \mathcal{P}(\mathbb{F})$ and a feature composition ordering $<$. The feature composition ordering is defined as a partial order $\{(f_1, f_2) \in \mathbb{F}^2\}$ which we write as $f_1 < f_2$. A valid feature configuration including $n$ features is a sequence of these features $[f_1, ..., f_n]$ such that for $i < j : f_i < f_j$.

To reason about the configurations that have a specific feature selected, we define the set of valid configurations for a distinct feature as follows.

**Definition 2.2** (Feature Configurations for Feature $f$)**.** Let $FM_f = \{fc \in FM \mid f \in fc\}$ be the set of valid configurations of feature model FM that contain feature $f$.

The variability that has been modeled on the domain level has to be implemented on the code level. As software product lines aim to implement a whole product family, features are implemented once and are then put together according to the selected feature configuration and ordering. There are many different techniques to realize this, but they generally separate into two different categories: *annotation-based* and *composition-based* mechanisms [5].

While annotation-based mechanisms provide a virtual separation of features, meaning that all features are implemented in a single code base, the composition-based ones offer a physical separation. Physical separation means that all artifacts belonging to a certain feature are modularized into one cohesive unit [5], called *feature module*. During product derivation, the units of all selected features in a feature configuration are composed. For this work, we prefer a

[1] https://github.com/TUBS-ISF/CorC/tree/VarCorCxFeatureIDE

composition-based mechanism, as it separates the implementation of one method into smaller refinements stored in the feature modules which offers a clearer structure compared to annotation-based mechanisms. The following definition of feature modules considers sets of methods as the implementation of a feature module. We generally declare all fields as public variables so that all methods have access to them.

**Definition 2.3** (Feature Module)**.** Let $\mathbb{M}$ be the universe of methods. Then we define the feature module of a feature $f$ as a set of methods $\text{impl}(f) \subseteq \mathbb{M}$. One method can be contained in more than one feature module of the product line.

**Feature-Oriented Programming.** In this work, we concentrate on *feature-oriented programming* [5, 9] as a specific composition-based technique. In feature-oriented programming, one method can be implemented in more than one feature module. Thereby, the different implementations of the method override each other in the feature composition ordering from smallest to greatest as defined in the feature model. However, there is the possibility to use an *original call* to call the implementation of the method in the smaller feature modules. This general mechanism is similar to overriding with super-calls in object-oriented languages like Java. We define the method refinement in feature-oriented programming as follows.

**Definition 2.4** (Method Refinement)**.** Let $\mathtt{m} \in \mathbb{M}$ be a method and $f_1, ..., f_n \in \mathbb{F}$ features occurring in the set of valid feature configurations of feature model FM and $\mathtt{m} \in \text{impl}(f_1), ..., \text{impl}(f_n)$. The feature composition ordering of FM is defined as $f_1 < ... < f_n$. Then the smallest feature containing $\mathtt{m}\,(\text{impl}(f_1))$ introduces the method. All other features *refine* this method by overriding it from smallest to greatest feature according to the feature composition ordering.

Based on this mechanism, the implementation of one method varies for different feature configurations. The distinct implementation for one configuration is formed by overriding the method in the feature modules of the features from the configurations from smallest to greatest and resolving the original calls with a call to the method implementation of the next smaller feature accordingly.

Besides the original call, which is resolved differently depending on the feature configuration, also normal method calls can have varying implementations. This is when a called method is defined in more than one feature module and therefore is also dependent on the distinct feature configuration. We use the term *variational method call* for a method call referring to a method with varying implementations.

## 2.2 Correctness-by-Construction

*Correctness-by-construction* [26] is an refinement-based, incremental approach[2] to create programs in the sense of total correctness. Every statement is surrounded by a specification forming a *Hoare triple* of the form $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$. Thereby, the precondition $\mathtt{P}$ marks the state of the program before the statement is executed and guarantees that the statement will terminate in the state described by the postcondition $\mathtt{Q}$. The precondition $\mathtt{P}$, the postcondition $\mathtt{Q}$, and the Hoare triple itself are *predicate formulas*, which means that they either evaluate to `true` or `false`. In this work, the pre- and postcondition are defined in *first-order logic* and the statement $\mathtt{S}$ is defined in *Guarded Command Language* [18], which uses the following five constructs: empty command (*skip*), assignment (:=), composition (;), selection (**if**), and repetition (**do**). In addition to these constructs, we allow to use method calls.

When developing a method with correctness-by-construction, the starting point is always a Hoare triple with an abstract statement. We also refer to the starting triple as the *contract* of that method.

**Definition 2.5** (Contract)**.** We define a contract $c$ in the form of $c = \{\mathtt{P}\}\mathtt{m}\{\mathtt{Q}\}$, with $\mathtt{P}$ being the precondition, $\mathtt{Q}$ the postcondition, and $\mathtt{m} \in \mathbb{M}$ the method. $\mathtt{P}$ and $\mathtt{Q}$ are both defined in first-order logic over all variables $v \in \mathbb{V}$. We require that each contract $c \in \mathbb{C}$ can be formulated as $c = \{\mathtt{P}\}\mathtt{m}\{\mathtt{Q}\}$ regardless of the particular specification language.

With the application of *refinement rules* that guarantee the correctness at each refinement step, the starting triple can then be refined to implement the method. Generally, the method is finished when no abstract statement is left. In Figure 1, we present a list of the six most important refinement rules and explain them in the following.

***Skip.*** Skip statements do not alter the program state [18, 26].

***Assignment.*** An abstract statement $\mathtt{S}$ can be replaced by an assignment $\mathtt{x} := \mathtt{E}$ if precondition $\mathtt{P}$ implies postcondition $\mathtt{Q}$ in which the variable $\mathtt{x}$ has been replaced by expression $\mathtt{E}$. We refer to this replacement with the notation $\mathtt{Q}[\mathtt{x} \backslash \mathtt{E}]$.

***Composition.*** The composition rule splits one abstract statement into two abstract statements $\mathtt{S}_1$ and $\mathtt{S}_2$. Additionally, an intermediate condition $\mathtt{M}$ has to be provided [18].

***Selection.*** The selection rule is used to refine the abstract statement differently in various cases that are defined by the guards $\mathtt{G_i}$. The Hoare triple is refined to $n$ more Hoare triples of the form $\{\mathtt{G_i} \land \mathtt{P}\}\mathtt{S_i}\{\mathtt{Q}\}$. The guards $\mathtt{G_i}$ are evaluated and the substatement of the first satisfied guard is executed.

***Repetition.*** Statement $\mathtt{S}$ is executed, as long as guard $\mathtt{G}$ evaluates to `true`. The repetition refinement rule additionally requires an invariant $\mathtt{I}$ and a variant $\mathtt{V}$. The termination of the repetition is verified by showing that the variant is monotonically decreasing with zero as the lower bound. Additionally, an invariant is needed to guarantee the postcondition.

[2] The term refinement has different meanings in feature-oriented programming and correctness-by-construction. When talking about the refinement of methods or a method being refined then we refer to Definition 2.4. For correctness-by-construction, we use the terms refinement rules or refinement steps and mean the refinement in the sense of the application of a refinement rule from Section 2.2 to concretize an abstract statement.

$\{P\}\ S\ \{Q\}$ *can be refined to*

1. *Skip* : $\{P\}$ *skip* $\{Q\}$ *iff* $P$ *implies* $Q$

2. *Assignment* : $\{P\}\ x := E\ \{Q\}$ *iff* $P$ *implies* $Q[x \backslash E]$

3. *Composition* : $\{P\}\ S_1\ ;\ S_2\ \{Q\}$ *iff there is an intermediate condition* $M$ *such that* $\{P\}\ S_1\ \{M\}$ *and* $\{M\}\ S_2\ \{Q\}$

4. *Selection* : $\{P\}$ **if** $G_1 \rightarrow S_1$ **elif** $\ldots\ G_n \rightarrow S_n$ **fi** $\{Q\}$ *iff* ($P$ *implies* $G_1 \lor G_2 \lor \ldots \lor G_n$) *and* $\{P \land G_i\}\ S_i\ \{Q\}$ *holds for all* $i$.

5. *Repetition* : $\{P\}$ **do** $[I, V]\ G \rightarrow S$ **od** $\{Q\}$ *iff* ($P$ *implies* $I$) *and* ($I \land \neg G$ *implies* $Q$) *and* *and* $\{I \land G \land V{=}V_0\}\ S\ \{I \land 0{\leq}V \land V{<}V_0\}$ *and* $\{I \land G\}\ S\ \{I\}$

6. *Method Call* : $\{P\}\ m(a_1, ..., a_n,\ b)\ \{Q\}$ *with method* $\{P'\}\ m(\text{param}\ p_1, ..., p_n,\ \text{return}\ r)\ \{Q'\}$ *iff* $P$ *implies* $P'[p_i \backslash a_i]$ *and* $Q'[p_i^{old} \backslash a_i^{old}, r \backslash b]$ *implies* $Q$

**Figure 1.** List of Refinement Rules in Correctness-by-Construction [26]

***Method Call.*** The method call refinement rule [26] introduces a method with contract and the following signature: $\{P'\}\ m(\text{param}\ p_1, ..., p_n, \text{return}\ r)\ \{Q'\}$.
Thereby, $p_1, ..., p_n$ represents a list of parameters, whose scope is limited to the method body, and $r$ represents a return variable that gets assigned a new value after the execution of method $m$. Both, $p_1, ..., p_n$ and $r$ can also be empty. To omit side effects in all cases, we define the parameters as call-by-value. As a result, the method call rule can generally be applied if the specification of the method complies with the specification of the statement. However, as the specification of method $m$ uses the formal parameters $p_1, ..., p_n$ and $r$ and the pre- and postcondition of the calling Hoare triple uses the actual parameters $a_1, ..., a_n$ and $b$ in $P$ and $Q$, the variables have to be replaced so that the implication is indeed provable. For the preconditions $P$ and $P'$ we replace the formal parameters $p_1, ..., p_n$ with the actual parameters $a_1, ..., a_n$ in $P'$ (i.e., $P'[p_i \backslash a_i]$). Because of the limited scope of the formal parameters, they are not allowed to appear in the postcondition $Q'$. However, $Q'$ may refer to their value before executing the method, which we denote as $p^{old}$. Therefore, we have to replace the original formal parameters by the original actual ones as well as the formal return parameter $r$ by the actual one $b$ ($Q'[p_i^{old} \backslash a_i^{old}, r \backslash b]$). The return value is not allowed to appear in the precondition, as it does not exist before executing the method.

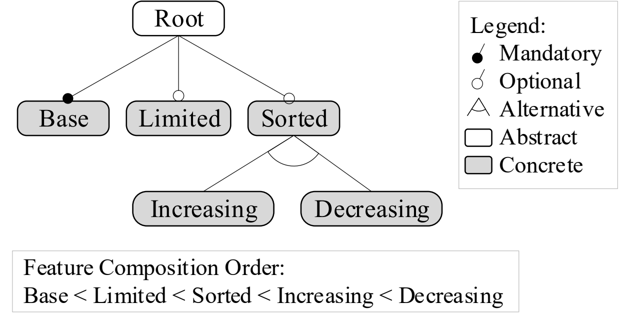


**Figure 2.** Feature Model of IntegerList Product Line

## 3 Motivating Example

In this section, we introduce the *IntegerList* as exemplary product line implementation. The *IntegerList* maintains an integer array `data` by offering corresponding methods. We show the feature model of the *IntegerList* product line in Figure 2. The method `push` adds a new integer `newTop` to the array in the implementation of feature *Base*. Feature *Limited* limits the length of `data` so that `newTop` is only added if the limit is not reached yet. Feature *Sorted* ensures that the array `data` is always sorted, either in an increasing or a decreasing order (sub-features *Increasing* and *Decreasing*). Even for this rather small example, there are in total six valid configurations (FM = {[*Base*], [*Base*, *Limited*], [*Base*, *Sorted*, *Increasing*], [*Base*, *Sorted*, *Decreasing*], [*Base*, *Limited*, *Sorted*, *Increasing*], [*Base*, *Limited*, *Sorted*, *Decreasing*]}) and the amount of possible configurations grows exponentially with the number of optional features. Using classical correctness-by-construction as introduced in the previous section, the developer would have to implement every method six times, i.e. once for every variant. Instead, we want to introduce a product line mechanism that enables the reuse of code. As mentioned in Section 2.1, we decided for feature-oriented programming, which offers the original call as a reuse mechanism.

In Figure 3, we give an overview of the development process using the *IntegerList* example. In the top left corner, there is the feature model that we just introduced. For every concrete feature in the feature model, a feature module is generated. In these feature modules the user can implement the features using the tool VarCorC, which implements the approach presented in this paper. Thereby, the cbc files in the feature modules correspond to one method implementation. In the example, the methods `push` and `sort` are contained in more than one feature module. Like in feature-oriented programming, they override each other in the feature composition order given by the feature model, but can use an additional correctness-by-construction refinement rule for the original call. As feature *Base* is the smallest feature according to the feature composition ordering the methods

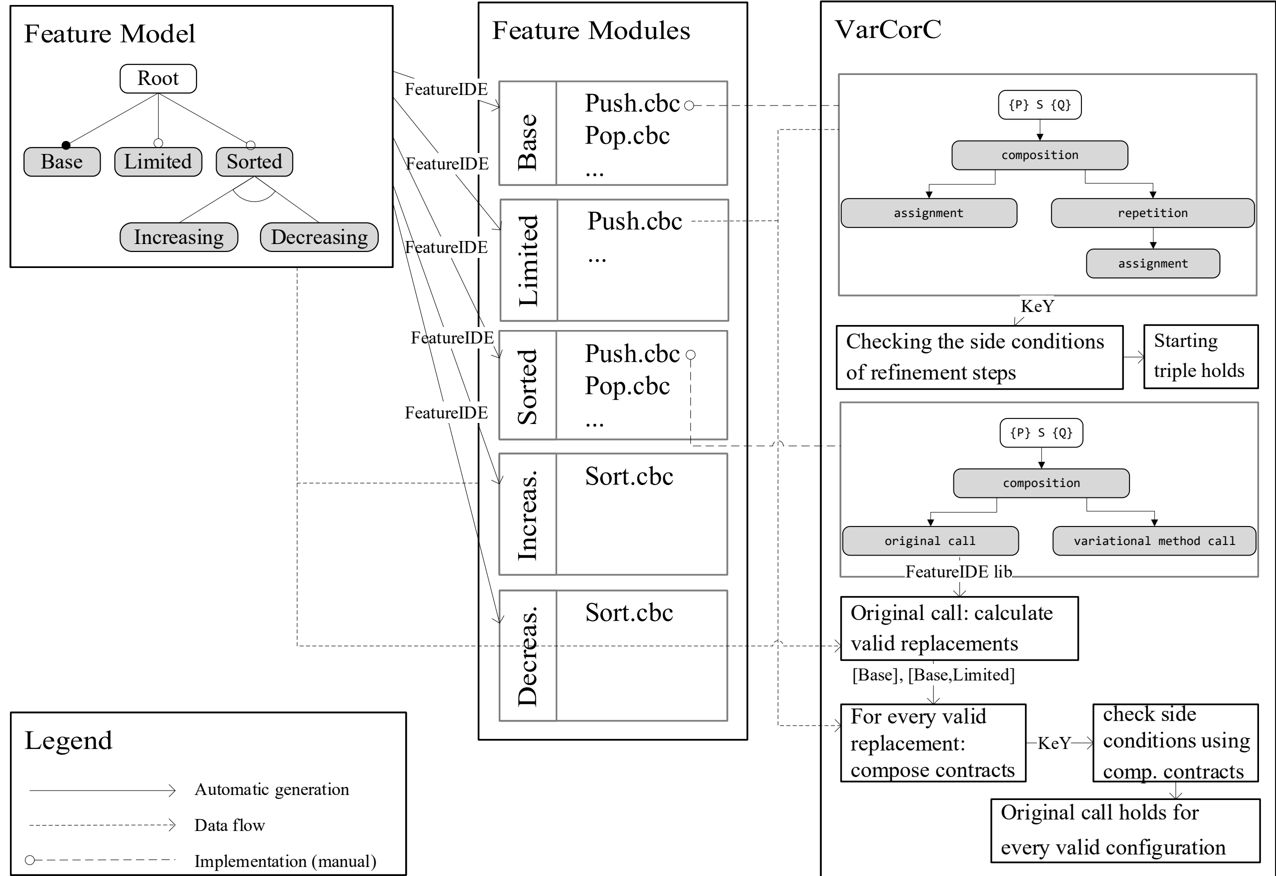


**Figure 3.** Process Overview

implemented by this feature are not allowed to use the original call. Therefore, methods in this feature module can be implemented using the standard set of refinement rules from Section 2.2 starting with a pre- and postcondition and an abstract statement. The developer can use the tool VarCorC to manually construct the method in a tree form similarly as shown in Figure 3. VarCorC can guarantee the correctness of the individual refinement steps performed by the developer and when no abstract statement is left the method can be considered as correct with respect to the pre- and postcondition from the starting Hoare triple.

In Figure 4, we show a more detailed version of method `push` in feature module *Sorted* from Figure 3. This example has also been used to motivate variational correctness-by-construction [10], however, we changed the third refinement step. Feature *Sorted* ensures that array `data` is always sorted. Therefore, it first uses an original call to reuse the implementation of the `push` method in a smaller feature according to the feature composition ordering, which adds the new element ②. Afterwards, it calls a method `sort` that sorts the array ③. As illustrated in Figure 3, to guarantee correctness for the original call the valid replacements need to be calculated first using the feature model and the information of which feature modules implement the method. In this example, the original call can either be replaced by the implementation of `push` in feature *Base* or a composition of *Base* and *Limited*. For the contract composition the pre- and postconditions from the starting triples of the `push.cbc` files from feature modules *Base* and *Limited* are needed. Afterwards, the side conditions of our proposed rule need to be checked to guarantee the correctness of this refinement step.

As seen in the example, there can be several replacements for an original call. Consequently, we need to define a *set of relevant replacements* that correctly captures all these method calls to guarantee the correctness of an original call. This set is defined by a particular context that regulates the use of the original call. In this paper, the context is feature-oriented software product lines, which restrict the valid replacements with the feature model and the definition of the original call, which only allows to call smaller features corresponding to the feature composition order.

An additional issue arises when looking at the call of method `sort` ③ in Figure 4. As seen in Figure 3, this method

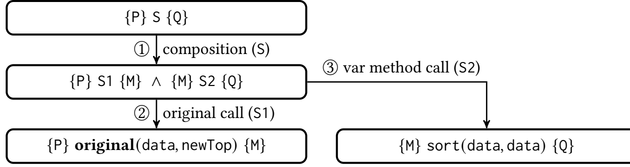


**Figure 4.** Refinement Steps of Method push in the Sorted Refinement, adapted from [10]

is defined in two feature modules, namely *Increasing* and *Deacreasing*. As the implementation of method sort changes due to the feature configuration, we refer to it as a *variational method call*. Similar to the original call, we have to establish side conditions for all relevant implementations of sort to guarantee correctness of the whole product line. In our example, method sort either refers to the implementation in feature *Increasing* or *Decreasing*, and both implementations need to be checked. As a result, we need to define a new refinement rule for variational method calls and the corresponding set of relevant replacements, which differs from the set for an original call. In summary, we can have two types of variation points in a method: an original call and a variational method call both of which can have different implementations depending on the feature configuration.

# 4 Applying Correctness-by-Construction to Software Product Lines

The key to guarantee the correctness for *original calls* and *variational method calls* is the flexible adaption of contracts. For this, we rely on contract composition which is explained in the following. Afterwards, we propose the original call refinement rule and the variational method call refinement rule to create correct-by-construction software product lines.

## 4.1 Contract Composition

To guarantee correctness for variation points that depend on implementations of different feature modules, we define the *context of a refinement rule*. This context serves as a basis to differentiate between different implementations of one method in several feature modules and therefore captures the method that is currently implemented, the feature the implementation belongs to, and the feature model which defines the dependencies of all features in the product line.

**Definition 4.1** (Context of a Refinement Rule)**.** We define the context of a refinement rule with the abstract statement S that is refined as a triple of the form context(S)=(FM, $f$, m) with feature model FM, feature $f$, and method m, which is implemented in the feature module impl($f$).

In feature-oriented programming, the developer creates different refinements of a method in different feature modules that can override each other with respect to the feature composition ordering. These method refinements can use the keyword original to call the implementation of the refinements that have been defined before, i.e., in feature modules that belong to features that are smaller according to the feature composition ordering. As this mechanism modifies methods and may also completely change a methods behavior in a way that violates the Liskov principle [39], it is crucial to be able to adapt their contracts as well. For example, when a new variant changes functionality of a method by refining it, the old contract of that method will most likely be insufficient to verify its behavior. In that case, the contract needs to be adapted as well using a specific composition technique. We define contract composition as follows.

**Definition 4.2** (Contract Composition)**.** Consider a contract composition mechanism M as a function $\bullet_{\mathrm{M}} : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ defined over the set $\mathbb{C}$ of all possible contracts. We take the original contract $c = \{\mathtt{P}\}\mathtt{m}\{\mathtt{Q}\}$, the refining contract $c' = \{\mathtt{P}'\}\mathtt{m}\{\mathtt{Q}'\}$, and define the composed contract as $c'' = c' \bullet_{\mathrm{M}} c = \{\mathtt{P}'\}\mathtt{m}\{\mathtt{Q}'\} \bullet_{\mathrm{M}} \{\mathtt{P}\}\mathtt{m}\{\mathtt{Q}\} = \{\mathtt{P}' \bullet_{\mathrm{M}} \mathtt{P}\}\mathtt{m}\{\mathtt{Q}' \bullet_{\mathrm{M}} \mathtt{Q}\} = \{\mathtt{P}''\}\mathtt{m}\{\mathtt{Q}''\}$. The concrete mechanisms for the contract composition define how $\mathtt{P}''$ and $\mathtt{Q}''$ are derived from $c$ and $c'$.

Note, that contract composition is always performed for one method m. As mentioned previously, in feature-oriented programming methods can be refined which results in different implementations in $c$ and $c'$. Nevertheless, the signature of the refined method is always maintained. Additionally, with correctness-by-construction, we only consider the pre- and postcondition and avoid the actual composition of the implementation, which is why we abstract from the differing implementations and only refer to method m in Defintion 4.2.

In previous work, three contract composition techniques that have been proposed by Thüm et al. [39] have already been adapted to compose contracts for variational correctness-by-construction [10]. In this work, we only use and explain one of them as it is sufficient to implement our running example. The mechanism presented in this paper is called *explicit contracting* and offers a keyword to explicitly refer to the contracts of the original method refinements, similar to the original call in feature-oriented programming.

**Example 4.3.** In Section 3, we already introduced the *IntegerList* product line and method push, that is implemented by the features *Base, Limited* and *Sorted*. In Listing 1, we additionally give the pre- and postconditions for method push in feature modules *Base* and *Sorted* and show the resulting composed contract for feature configuration $fc = [Base, Sorted]$. In the following, we will describe the contract composition for the postcondition of method push. The precondition can be composed analogously. The postcondition of method push in feature *Base* expresses that the new element newTop has been added to array data and that all elements that have been stored in data are still contained after the execution of method push. Consequently, the postcondition can be formulated as seen in Listing 1. Thereby, the first predicate, contains(int[] a, int elem), evaluates to true if array

```
context(S) = (FM,Base,push)                                  Base
P_Base = data != null
Q_Base = contains(data, newTop) & containsAll(data^old
    , data)
```

```
context(S) = (FM,Sorted,push)                                Sorted
P_Sorted = \original
Q_Sorted = \original & isSorted(data)
```

```
P = data != null                                      Base • Sorted
Q = (contains(data, newTop) & containsAll(data,
    data^old)) & (isSorted(data))
```

**Listing 1.** Pre- and Postcondition of Method push in Feature Modules impl(*Base*) and impl(*Sorted*) and Composed Contract for push with $fc = [Base, Sorted]$

```
context(S) = (FM,Limited,push)                               Limited
P_Limited = \original
Q_Limited = (data^old.length < LIMIT) → \original
```

```
P_Base•Limited = data != null                        Base • Limited
Q_Base•Limited = (data^old.length < LIMIT) → (contains(
    data, newTop) & containsAll(data^old, data))
```

```
P = data != null                           Base • Limited • Sorted
Q = ((data^old.length < LIMIT) → (contains(data,
    newTop) & containsAll(data, data^old))) & (
    isSorted(data))
```

**Listing 2.** Pre- and Postcondition of Method push in Feature Module impl(*Limited*) and Composed Contract for Method push with $fc = [Base, Limited]$ and $fc = [Base, Limited, Sorted]$

a contains the integer `elem` and to `false` otherwise. The second predicate, `containsAll(int[] a, int[] b)`, evaluates to `true` if array b contains all elements that are contained in array a and `false` otherwise. Feature *Sorted* refines the push method such that the array remains sorted (cf. Section 3). As the contract should reflect the behavior of method push, we want to add a predicate `isSorted(int[] a)` to its postcondition if feature *Sorted* has been selected. We define the postcondition of the push method refinement in the feature module of feature *Sorted* as $Q_{Sorted}$ = `\original & isSorted(data)` with `\original` being resolved to $Q_{Base}$.

In product lines it is common to refine a method more than once. This results in *sequences of contract compositions*. A sequence consists of features from a valid feature configuration and results in one composed contract for one method. For example, a method `a()` may have a different refinement sequence than method `b()` for the same feature configuration, as the methods might be implemented by different feature modules from the configuration. We formally define a contract composition sequence as follows.

**Definition 4.4** (Contract Composition Sequence)**.** We define a *contract composition sequence* as the contract $c$ resulting from the contract composition of a sequence of $n$ features $[f_1, ..., f_n]$. The composition is performed methodwise. Let $c_k$ be the contract of method m which is contained in one or more feature modules $m \in impl(f_i)$ with $i \in \{1, ..., n\}$, $k \in \{1, ..., l\}$, *and* $l \leq n$, then $c$ is composed as

$$c = c_1 \bullet_{M2} c_2 \bullet_{M3} \ldots \bullet_{Ml} c_l$$

resulting in a contract $c$ of the form described in Definition 2.5. The contract composition mechanisms $\bullet_{M2}$ to $\bullet_{Ml}$ have to be a function as described in Definition 4.2 and are defined by the greater feature module according to the feature composition ordering.

It is important to notice that method m does not necessarily have to be contained in every feature module of the sequence $[f_1, ..., f_n]$. Therefore, the number of composed contracts $l$ can be less or equal than the number of features $n$. Furthermore, in the following we will use the short form • to refer the contract composition with the mechanism *explicit contracting*, as it is the only mechanism we use here.

**Example 4.5.** In Listing 2, we give the pre- and postcondition of method push in feature module impl(*Limited*) and show the composed contract for feature configuration $fc = [Base, Limited, Sorted]$ with an intermediate result in the middle box. The contract composition sequence for feature configuration $fc = [Base, Limited, Sorted]$ in the *IntegerList* product line of method push is defined as: $c = c_{Base} \bullet c_{Limited} \bullet c_{Sorted}$. The contract composition for postcondition Q of the contract of the push method starts with the composition of *Base* and *Limited*. Therefore, the keyword `\original` is replaced with the postcondition of feature *Base*. In the next step, the resulting postcondition $Q_{Base \bullet Limited}$ is composed with the postcondition of feature *Sorted*:

$$Q = Q_{Base \bullet Limited} \bullet Q_{Sorted}$$

## 4.2 Original Call

In this subsection, we extend correctness-by-construction with a refinement rule for original calls, which is the first variation point that can be used in feature-oriented software product lines. The original call refinement rule is based on the method call refinement rule introduced earlier in Section 2.2. To guarantee the correctness for an original call, the contracts of all relevant replacements have to comply with the contract of the calling statement. In this paper, the set of relevant replacements is given by the feature model, which explicitly defines dependencies between features and the order in which they are composed. Consequently, we need to define a set of *relevant feature sequences* for the original call refinement rule according to these specific restrictions to guarantee correctness for the whole product line. The

following definition for the set of relevant feature sequences for original calls is defined by three insights:

1. Only feature configurations involving feature $f$ from context(S) can contain relevant replacements for the original call.
2. Valid replacements can only be formed by features that are smaller than feature $f$ which is a consequence of the feature composition ordering from the feature model FM. It determines the order in which the features are composed, or in other words, which features can actually be called by the original call.
3. Features that do not implement method m from context(S) do not alter the resulting composed contract.

**Definition 4.6** (Relevant Feature Sequences for Original Call)**.** Let the context of the abstract statement S be context(S)=(FM, $f$, m). The set of features of a given configuration $fc$ that are smaller than feature $f$ in the feature composition ordering of FM and implement method m is defined by $fc_{\mathtt{m}} = \{f' \in fc \mid \mathtt{m} \in \text{impl}(f') \textit{ and } f' < f\}$.

Then, the set of configurations of the feature model FM containing feature $f$ projected onto the smaller features implementing method m is defined as $\text{FM}_f^{\mathtt{m}} = \{fc_{\mathtt{m}} \mid fc \in \text{FM}_f\}$.

We define the set of *relevant feature sequences for an original call* in the context(S) as

$$c_f^{\mathtt{m}} = \{[fc_f^{\mathtt{m}}] \mid fc_f^{\mathtt{m}} \in \text{FM}_f^{\mathtt{m}}\}$$

We define the original call refinement rule based on Definition 4.6 for the set of relevant feature sequences as follows:

**Definition 4.7** (Original Call Refinement Rule)**.** Let the context of the abstract statement S be context(S)=(FM, $f$, m). Then {P} S {Q} can be refined to {P} original($\mathtt{a}_1, ..., \mathtt{a}_n$, b) {Q} iff for all contract composition sequences $c_i$

$$c_i = \{\mathtt{P}_i\}\ \mathtt{m}(\mathtt{param}\ \mathtt{p}_1, .., \mathtt{p}_n,\ \mathtt{return}\ \mathtt{r})\ \{\mathtt{Q}_i\}$$

for the relevant feature sequences $c_f^{\mathtt{m}}$ as defined in Definition 4.6, it holds that

$$\mathtt{P}\ \textit{implies}\ \mathtt{P}_i[\mathtt{p_j} \backslash \mathtt{a_j}]\ \textit{and}\ \mathtt{Q}_i[\mathtt{p_j^{old}} \backslash \mathtt{a_j^{old}}, \mathtt{r} \backslash \mathtt{b}]\ \textit{implies}\ \mathtt{Q}$$

**Example 4.8.** In Figure 4, we already displayed the implementation of method push from our previous example in feature module impl(*Sorted*) with an original call refinement rule in step ②. The context is defined as context(S1) = (FM, *Sorted*, push) and the resulting set of relevant feature sequences $c_{Sorted}^{\mathtt{push}} = \{[Base], [Base, Limited]\}$. These feature sequences need to be considered when applying the original call refinement rule. The pre- and postconditions $\mathtt{P}_i$ and $\mathtt{Q}_i$ are retrieved using contract composition over the relevant feature sequences as described in Section 4.1.

### 4.3 Variational Method Call

The second type of variation point that can be used in software product lines is the call of a method that can have different implementations due to its definition in the product line, i.e., a *variational method call*. We call a method variational if it is defined in more than one feature module. However, it is not necessary that an original call is used, because overriding a method also results in a different implementation. We define a variational method as follows.

**Definition 4.9** (Variational Method)**.** We define the term *variational method* to describe a method varM that is defined in more than one feature module. In other words, there are at least two features $f_i \in \mathbb{F}$, $i \in \{1, \ldots, n\}$ such that varM $\in$ impl($f_i$), and for all these $f_i$ there is at least one configuration $fc \in \text{FM}$ *such that* $f_i \in fc$.

Similar to an original call, the implementation and contract of a variational method also depends on the feature configuration. Therefore, to guarantee correctness for the whole product line, we define the *set of relevant feature sequences for variational method calls* as well. This set is defined by two insights:

1. Only feature configurations that involve feature $f$ from context(S) can contain relevant replacements for a variational method call.
2. Features that do not implement method varM do not alter the resulting composed contract.

**Definition 4.10** (Relevant Feature Sequences for Variational Method Call)**.** Let the context of the abstract statement S be context(S)=(FM, $f$, m), and let method varM be a variational method that is called by method m. Method varM is defined in the feature module impl($f'$) for $f' \in fc$ *and* $fc \in \text{FM}_f$ ($\mathtt{varM} \in \{impl(f') \mid f' \in fc\}\ for\ fc \in \text{FM}_f$).

The set of features of a given configuration $fc$ implementing varM is defined by $fc_{\mathtt{varM}} = \{f'' \in fc \mid \mathtt{varM} \in \text{impl}(f'')\}$.

Then, the set $\text{FM}_f^{\mathtt{varM}}$ is the set of configurations of a feature model FM containing feature $f$ projected onto the features implementing varM is defined as $\text{FM}_f^{\mathtt{varM}} = \{fc_{\mathtt{varM}} \mid fc \in \text{FM}_f\}$. We define the set of *relevant feature sequences for a variational method call* in the context(S) as

$$c_f^{\mathtt{varM}} = \{[fc_f^{\mathtt{varM}}] \mid fc_f^{\mathtt{varM}} \in \text{FM}_f^{\mathtt{varM}}\}$$

The first insight above is the same as for original calls. The second insight above is very similar to the third insight for original calls with the difference that in this case only features that implement the called method varM are important instead of method m from the context(S). Consequently, feature $f$ from context(S) is not necessarily contained in the resulting feature sequences from the set of relevant feature sequences, if feature $f$ does not implement method varM. Another difference to the set of relevant feature sequences for original calls is that all features, no matter if they are smaller or greater than feature $f$, can be part of the valid replacements. This is due to the fact that the refinement sequence of the called method varM is independent of the refinement sequence of method m from context(S). That method varM

exists and can be called has to be guaranteed by the feature model and is orthogonal to the problem considered here.

The variational method call refinement rule can be defined analogously to the original call refinement rule, but uses a different set of relevant feature sequences. We define the variational method call refinement rule using the set of relevant feature sequences from Definition 4.10 as follows.

**Definition 4.11** (Variational Method Call Refinement Rule)**.** Let the context of the abstract statement S be context(S) = (FM, *f*, m). Then {P} S {Q} *can be refined to* {P} varM($a_1$, ..., $a_n$, b) {Q} *iff* for all contract composition sequences $c_i$

$$c_i\text{=}\{P_i\}\ \texttt{varM}(\texttt{param}\ p_1, .., p_n,\ \texttt{return}\ r)\ \{Q_i\}$$

of the relevant feature sequences $c_f^{\texttt{varM}}$ from Definition 4.10, it holds that

$$P\ \textit{implies}\ P_i[p_j \backslash a_j]\ \textit{and}\ Q_i[p_j^{old} \backslash a_j^{old}, r \backslash b]\ \textit{implies}\ Q$$

**Example 4.12.** To showcase a variational method call, we use our previous example of the *IntegerList* product line from Section 3 again. We displayed the implementation of method push in feature *Sorted* in Figure 4 with a method call of method sort in step ③. As it is defined by two features, namely *Increasing* and *Decreasing*, sort is a variational method. The context is defined as context(S2) = (FM, *Sorted*, push) and the resulting set of relevant feature sequences $c_{Sorted}^{\texttt{sort}} = \{[Increasing], [Decreasing]\}$. These feature sequences need to be considered when applying the variational method call refinement rule.

### 4.4 Proof of Soundness

In this subsection, we reason about the soundness of our approach which means that all programs in a product line are correct if the product line is constructed using our refinement rules. Concretely, we consider this on a per-method basis meaning that the product line is correct if all methods that can be generated for all valid feature configurations satisfy their pre- and postconditions. We assume that the refinement rules from Figure 1 are sound [28]. Hence, for soundness we only need to consider the refinement rules introduced in Section 4.2 and 4.3.

**Theorem 4.13** (Soundness)**.** *Let* FM *be a feature model, f a feature, and* m *a method implemented in* impl(*f*). *If the starting triple of method* m {P}S{Q} *with abstract statement S and* context(S) = (FM, *f*, m) *is refined to* {P}C{Q} *with a concrete implementation* C *following the refinement rules in Figure 1 and the rules that have been proposed in Section 4.2 or Section 4.3, then for all valid feature configurations fc of feature model* FM *which contain the method* m, {P}C{Q} *holds.*

*Proof.* We show the above result by structural induction over the refinement rules. We assume that the rules presented in Section 2.2 already satisfy the result. Hence, it suffices to consider the original call and variational call rules introduced in Section 4.

- *Original Call Rule:* By applying the original call refinement rule, an abstract statement S is refined to {P}original(){Q}. The original() call is a placeholder for a method call that resolved via the set of relevant feature sequences determined by context(S). By definition, the set of relevant feature sequences contains all possible replacements for the original call and only those. Hence, we can apply the method call refinement rule for each element of the relevant feature sequences. For the method call rule, the soundness result already holds, which closes this case of the proof.
- *Variational Method Call Rule:* By applying the variational method call refinement rule, an abstract statement S is refined to {P}varM(){Q}. Thereby, varM() is a method call that can have different implementations which are resolved by the set of relevant feature sequences as defined in Definition 4.10. By definition, the set of relevant feature sequences contains all possible replacements for the variational method call and only those. Hence, we can apply the method call refinement rule for each element of the relevant feature sequences. For the method call rule, the soundness result already holds, which closes this case of the proof. □

### 4.5 Discussion

***Similarity of the Refinement Rules.*** Our proposed refinement rules from Definition 4.7 and 4.11 are defined similarly. In fact, the only differences are the replacement of the abstract statement S to either original or varM and the set of relevant feature sequences they use. Consequently, they could be defined as one refinement rule capturing both cases. However, we decided to explicitly differentiate between these two cases, as they generally underlie different concepts.

Comparing our proposed refinement rules to the method call refinement rule from Section 2.2, they all use the same condition to determine whether an abstract statement S can be refined to either a method call, variational method call, or original call. Indeed, the variational method call refinement rule is equal to the basic method call refinement rule when the set of relevant feature sequences would only consist of one feature configuration containing one feature, i.e., if the definition of the method was not variational. As a result, one could not only define our proposed refinement rules in one refinement rule, but also include the basic method call refinement rule.

***Exponential Explosion.*** Our proposed refinement rules and the corresponding sets of relevant feature sequences generate and establish the side conditions of all possible method variants to guarantee the correctness of the whole product line. Therefore, our approach can be categorized as product-based analysis strategy, which means the analysis of

each (method) variant in isolation [37], even if we technically do not verify complete variants of the product line. We are aware of the fact that this does not scale to larger product lines due to a combinatorial explosion of possible variants. To address this problem, other strategies for product line analysis, e.g., family-based strategies, have been proposed, which leverage reuse to reduce verification effort [6, 33, 37]. As our approach is the first one that extends correctness-by-construction for software product line development it can be seen as baseline for the application of those advanced strategies, which we plan to experiment with in future.

***Formal Verification.*** As with all formal verification techniques, our approach strongly depends on the quality of the formal specification given by the developer. There will always be a correlation between specification and the proof. In other words, the weaker the specification, the weaker is also the guarantees provided by the proof and vice versa. Additionally, we can only guarantee the correctness of the program with respect its specification and not the correctness of the specification itself.

## 5 Evaluation

In this section, we provide an evaluation focusing on feasibility. We implemented our approach in a tool called VarCorC, which we introduce first. Afterwards, we describe the settings and present and discuss the results.

### 5.1 Tool Support: VarCorC

We extended the tool VarCorC to enable the development of feature-oriented software product lines using the original call and variational method call refinement rule. VarCorC[3] is an open-source Eclipse plugin that already implemented variational correctness-by-construction [10] and is based on the tool CorC [34]. At its core, it has a correctness-by-construction meta-model modeled with the Eclipse Modeling Framework (EMF).[4] It offers a graphical editor which uses the underlying meta-model. The graphical editor is implemented with Graphiti.[5] It visualizes the refinement steps in a tree structure similar to Figure 4 starting with one Hoare triple at the top that can be refined as desired. Thereby, the pre- and postcondition from the top are propagated automatically to the refinements. To prove the correctness of the refined Hoare triples, the deductive verification tool KeY [4] is used.

To support the development of software product lines, we integrated the FeatureIDE library into VarCorC. FeatureIDE [38] is a common framework that supports software product line engineering. It offers a project structure already including a feature model similar to Figure 2, which can be designed in the feature model editor of FeatureIDE. As we decided for feature-oriented programming in our approach, we added folders representing feature modules and contain methods created in VarCorC. Using the FeatureIDE library, we are then able to calculate the sets of relevant feature sequences and use them to check the side conditions of the original call and variational method call refinement rule.

In Figure 5, we show a screenshot of VarCorC. On the left, there is the project explorer showing the structure of the project. At the bottom, there is a file called `model.xml` containing the feature model of the product line. In folder *features*, there are the feature modules named after the features in the feature model. They contain the implementation as VarCorC programs in folder *diagram* and corresponding proof files in a folder called *prove<Name>*. In folder *src*, Java classes can be added that can be used in VarCorC programs. Lastly, folder *src-gen* is used to compose and save contracts.

In the center of Figure 5, there is the VarCorC-diagram of method `push` in feature module *Sorted* (`push.diagram`) that we already used as an example throughout this paper. On the right, there are two blocks to define the variables that are used in the diagram and global conditions that always have to be fulfilled. The blocks in the center are connected with arrows resembling the refinements to the actual program similarly as shown in Figure 4. The information that is designed in this graphical representation is saved in an instance of the correctness-by-construction EMF model (`push.cbcmodel`).

The program can be created using the correctness-by-construction refinement rules that we presented in this paper. The user always starts with a box called *Formula* as the root node and can then create other boxes for compositions, selections, statements, etc. and connect them with the arrows resembling the refinements. After creating this tree structure, the boxes have red edges. To check the correctness of the created program, the user can trigger a verification process by right clicking on one of the boxes. This verification process translates the Hoare triple of the selected box into a key file using the corresponding refinement rule. If KeY is then able to verify the generated side conditions, there is an output in the console and the edges of that box become green. In the screenshot, the verification process has been triggered for the box in the right bottom corner which calls the method `sort`. As this is a variational method, the variational method refinement rule is applied. Therefore, the relevant feature sequences are calculated from the feature model in `model.xml` first. Afterwards, the contracts are composed and the side conditions are verified for the two relevant feature sequences, namely [*Increasing*] and [*Decreasing*].

In contrast to our approach presented in this work, variational correctness-by-construction [10] is a mechanism to create single variational methods. These methods are treated as standalone variational algorithms rather than in the context of a complete software product line. This means two things. First, there is no global definition of the variability in any form. The original call is used as a placeholder for composed methods which do not rely on the global constraints

[3] https://github.com/TUBS-ISF/CorC/tree/VarCorCxFeatureIDE
[4] https://eclipse.org/emf/
[5] https://eclipse.org/graphiti/

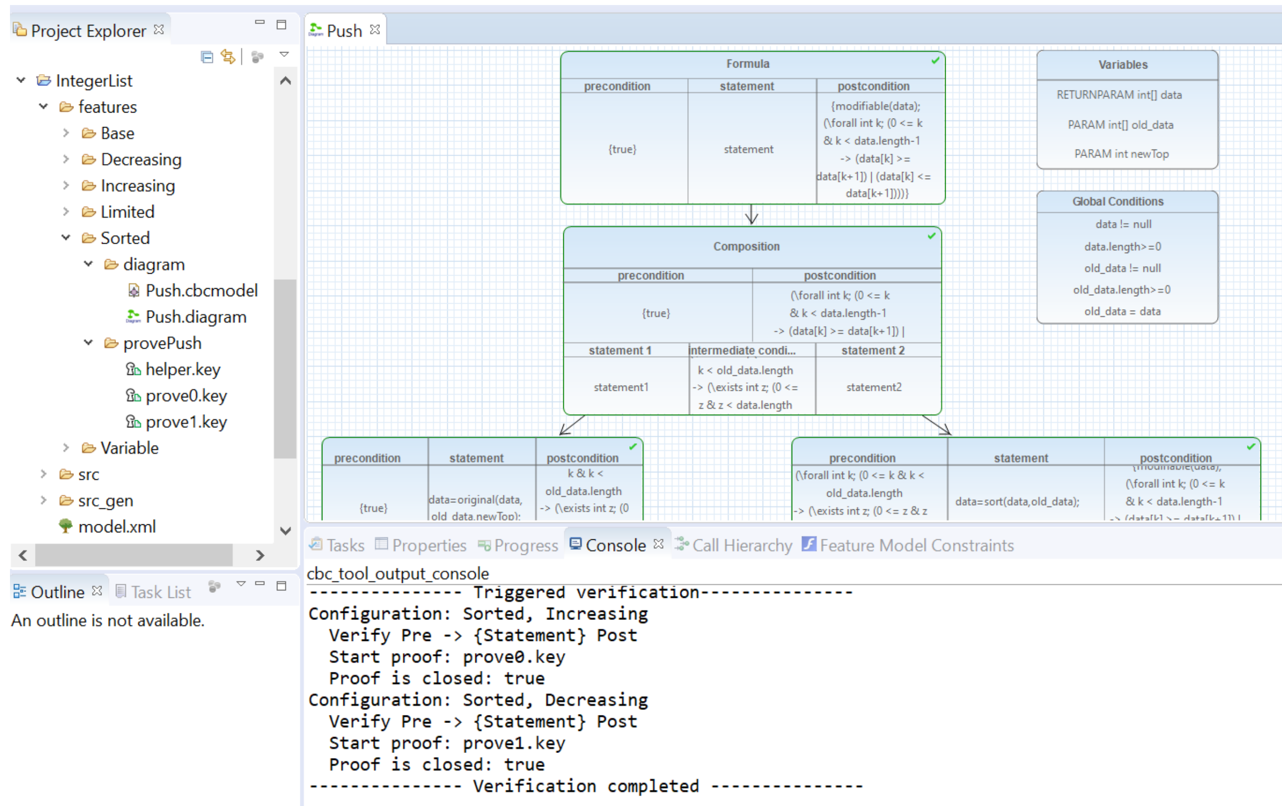


**Figure 5.** VarCorC-Diagram of the push Method in Feature Module *Sorted*

given by the feature model. Ergo, the user has to manually define which methods could possibly replace an original call for every method refinement. This manual process is tedious and error-prone compared to an automatic reasoning based on a feature model. Second, the correctness for the interaction between two variational methods cannot be guaranteed. This results from the missing global definition of the features because the caller does not have any information about the possible compositions of the called method.

### 5.2 Experimental Setup

As our approach is the first one to combine software product lines and correctness-by-construction, we focus on *feasibility* in our evaluation. We applied our approach to two prominent product line case studies, namely *IntegerList* and *BankAccount*. Both were already used (1) for *specifying* software product lines [39] and (2) as case studies for CorC [34] and VarCorC [10]. This gives us insights to what extent our approach can be used to create correct product lines.

The *IntegerList* product line has been used as a running example in Section 3 and 4. It resembles a class `IntList` which maintains an integer array and offers methods to add or remove elements from it. Method `push` inserts elements into the array and is implemented with three refinements. Additionally, we added the features *Increasing* and *Decreasing* which refine method `sort`. The *BankAccount* product line provides its core functionality in class `Account`, which comprises a total of four variational methods to manage a bank account. Thereby, the methods `update` and `undoUpdate` manipulate the savings of the account, and `nextDay` and `nextYear` allow to reason about withdrawal limits of users. For both case examples, we used the structure of FeatureIDE projects which means we included a feature model and feature modules encapsulating the programs created with VarCorC.

### 5.3 Experimental Results

In our experiment, we manually created 14 method refinements in VarCorC, one for each method and corresponding feature module of the two case studies. Two of the programs have been created from scratch and twelve could be adapted from former case studies [10]. VarCorC supports original calls as in feature-oriented programming (cf. Section 4.2), calls to other variational methods (cf. Section 4.3), and the composition of specifications using one of three contract composition techniques (cf. Section 4.1). Thereby, the relevant feature sequences for the proof of the original call and variational method call refinement rules are retrieved automatically from the feature model. In total, we could derive 16 different method variants with respect to valid feature configurations of the feature model. This includes

seven original calls and one variational method call. As expected, all proofs passed for all 16 variants, which means that guarantees were preserved during feature composition. We conclude that, with VarCorC, it is possible to develop correct-by-construction software product lines.

## 6 Related Work

In this paper, we address correctness-by-construction for obtaining provably correct software product lines. Therefore, we present related work on *post-hoc verification techniques* for product lines and other *refinement-based approaches*.

***Verification Techniques for Software Product Lines.*** To the best of our knowledge, there is only one other work regarding the development of software product lines being phrased as correctness-by-construction, which has been proposed by Pham [31]. The authors generate correct-by-construction product variants from a product line by composing proof artifacts that have been created by a post-hoc verification framework. Hence, this idea of correctness-by-construction is rather the assembly of post-hoc proof artifacts than extending the refinement rules of correctness-by-construction itself, as we did. Thüm et al. [40] also proposed an approach for proof composition for feature-oriented product lines written in Java. They used the Java Modeling Language (JML) for specification and composed proof obligations for the proof assistant Coq [16].

All other verification techniques for software product lines rely on post-hoc verification [37] which is a difference to the incremental approach of stepwise refinement that we apply by using correctness-by-construction. The following verification techniques all apply their mechanisms to design-by-contract contracts based on JML and use post-hoc verification for the proofs. Thüm et al. [39] defined six *contract composition techniques* and applied them to software product lines implemented with Java. We adapted three of these composition techniques to form the contracts for the original calls and the variational method calls. Bruns et al. [11] propose a mechanism called delta-oriented slicing that is similar to contract overriding, but adds the removal of contracts in delta modules. Hähnle and Schaefer [23] propose another composition technique for delta-oriented programming, which is a restrictive form of explicit contracting.

Gonçalves et al. [21] proposed a model-based approach for dataflow programs with specifications which can be transformed to a concrete implementation. Additionally, they apply feature labels to components of the dataflow program to implement software product lines in an annotation-based way. The general mechanism is related to correctness-by-construction as it is also refinement-based, however the main difference to our approach is that the focus of their work is on dataflow programs, our approach works on code-level rather than on model-level, and their product line mechanism is annotation-based while ours is composition-based.

***Refinement-based Verification.*** The Event-B framework [2] uses automata-based systems with specifications which are refined to a concrete implementation. Therefore, it is related to correctness-by-construction. There is also tool support for the Event-B framework. Atelier B [1] implements the B method by providing an automatic and interactive prover and Rodin [3] implements the Event-B method, which is considered an evolution of the B method. However, the main difference to VarCorC is that Atelier B and Rodin do not implement variability and work on automata-based systems rather than on code and specifications.

ArcAngel [29] is a tool that implements a tactic language for refinements to apply a sequence of rules based on Morgan's refinement calculus. These rules are applied to an initial specification to generate a correct implementation in the end. Unlike VarCorC, ArcAngel does not offer a graphical editor to visualize the refinement steps. Another difference is that ArcAngel creates a list of proof obligations that have to be proven separately. CRefine [30] is another related tool for the Circus refinement calculus which is a calculus for state-rich reactive systems. It also provides a GUI for the refinement process. The difference is that they use a state-based language and VarCorC uses code and specifications. ArcAngelC [15] extends CRefine by adding refinement tactics. Back et al. [7, 8] build the invariant-based programming tool SOCOS. They start explicitly with the specfication of not only pre- and postconditions, like it is done using correctness-by-construction, but also invariants before the coding process. However, none of the above approaches is applied to software product lines

## 7 Conclusion

Software product lines are increasingly developed for safety-critical systems, which makes their behavioral correctness an important concern. In this paper, we proposed an approach to develop correct-by-construction software product lines as an alternative to post-hoc verification to guarantee their correctness. Therefore, we defined two refinement rules extending correctness-by-construction that capture the two variation points that can be used in a method in feature-oriented programming: original calls and variational method calls. Furthermore, we proved their soundness and showed feasibility by using the tool VarCorC to conduct two case studies. Thereby, we were able to propose the first approach to develop feature-oriented software product lines using correctness-by-construction which will serve as a baseline for further extensions and improvements. Based on our results, we plan on further improvements by applying advanced product line verification strategies to increase efficiency and overcome limitations due to the combinatorial explosion of possible variants.

## References


[1] Jean-Raymond Abrial. 2005. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press.

[2] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering* (1st ed.).

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12, 6 (2010), 447–466.

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification – The KeY Book.*

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.*

[6] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-Interaction Detection Based on Feature-Based Specifications. 57, 12 (2013), 2399–2409.

[7] Ralph-Johan Back. 2009. Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects of Computing* 21, 3 (2009), 227–244.

[8] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. 2007. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *International Conference on Tests and Proofs.* Springer, 61–78.

[9] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. 30, 6 (2004), 355–371.

[10] Tabea Bordis, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. 2020. Variational Correctness-by-Construction. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems.* 1–9.

[11] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. 2011. Verification of Software Product Lines with Delta-Oriented Slicing. 61–75.

[12] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 16–30.

[13] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic Model Checking of Software Product Lines. 321–330.

[14] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.*

[15] Madiel Conserva Filho and Marcel Vinicius Medeiros Oliveira. 2012. Implementing Tactics of Refinement in CRefine. In *International Conference on Software Engineering and Formal Methods.* Springer, 342–351.

[16] Coq Development Team. 2010. *The Coq Proof Assistant Reference Manual.* LogiCal Project. Version 8.3.

[17] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.*

[18] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. 18, 8 (1975), 453–457.

[19] Edsger W. Dijkstra. 1976. *A Discipline of Programming* (1st ed.). Prentice Hall PTR.

[20] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. 391–400.

[21] Rui C Gonçalves, Don Batory, Joao L Sobral, and Taylor L Riché. 2017. From Software Extensions to Product Lines of Dataflow Programs. *Software & Systems Modeling* 16, 4 (2017), 929–947.

[22] David Gries. 1981. *The Science of Programming* (1st ed.).

[23] Reiner Hähnle and Ina Schaefer. 2012. A Liskov Principle for Delta-Oriented Programming. 32–46.

[24] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[25] Christian Kästner and Sven Apel. 2008. Type-Checking Software Product Lines—A Formal Approach. 258–267.

[26] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-by-Construction Approach to Programming.*

[27] Jing Liu, Josh Dehlinger, and Robyn Lutz. 2007. Safety Analysis of Software Product Lines Using State-Based Modeling. 80, 11 (2007), 1879–1892.

[28] Carroll Morgan. 1998. *Programming from Specifications.* Prentice Hall.

[29] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. 2003. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing* 15, 1 (2003), 28–47.

[30] Marcel Vinicius Medeiros Oliveira, Alessandro Cavalcante Gurgel, and CG Castro. 2008. CRefine: Support for the Circus Refinement Calculus. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods.* IEEE, 281–290.

[31] Thi-Kim-Dung Pham. 2017. *Development of Correct-by-Construction Software using Product Lines.* Ph.D. Dissertation. Paris, CNAM.

[32] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.*

[33] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. 347–350.

[34] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. 25–42.

[35] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. 77–91.

[36] Thomas Thüm. 2015. *Product-Line Specification and Verification with Feature-Oriented Contracts.* Ph.D. Dissertation. University of Magdeburg.

[37] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. 47, 1 (2014), 6:1–6:45.

[38] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. 79, 0 (2014), 70–85.

[39] Thomas Thüm, Alexander Knüppel, Stefan Krüger, Stefanie Bolle, and Ina Schaefer. 2019. Feature-Oriented Contract Composition. 152 (2019), 83–107.

[40] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. 2011. Proof Composition for Deductive Verification of Software Product Lines. 270–277.