# Variational Correctness-by-Construction

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

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

Alexander Knüppel
a.knueppel@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Thomas Thüm
thomas.thuem@uni-ulm.de
University of Ulm
Ulm, Germany

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

## ABSTRACT

Nowadays, the requirements for software and therefore also the required complexity is increasing steadily. Consequently, various techniques to handle the growing demand for *software variants* in one specific domain are used. These techniques often rely on variable code structures to implement a whole product family more efficiently. Variational software is also increasingly used for safety-critical systems, which need to be verified to guarantee their functionality in-field. However, usual verification techniques can not directly be applied to the variable code structures of most techniques. In this paper, we propose *variational correctness-by-construction* as a methodology to implement variational software extending the correctness-by-construction approach. Correctness-by-construction is an incremental approach to create and verify programs using small tractable refinement steps guided by a specification following the design-by-contract paradigm. Our contribution is threefold. First, we extend the list of refinement rules to enable variability in programs developed with correctness-by-construction. Second, we motivate the need for contract composition of refined method contracts and illustrate how this can be achieved. Third, we implement variational correctness-by-construction in a tool called VarCorC. We successfully conducted two case studies showing the applicability of VarCorC and were able to assess reduced verification costs compared to post-hoc verification as well.

## KEYWORDS

correctness-by-construction, deductive verification, formal methods, design-by-contract, variational software

**ACM Reference Format:**
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 (VaMoS '20), February 5–7, 2020, Magdeburg, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377038


*VaMoS '20, February 5–7, 2020, Magdeburg, Germany*

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377038

## 1 INTRODUCTION

It has become common practice in industry to develop different *variants* of a software to meet the individual needs of different customers [9]. New variants may add, remove, or refine functionality compared to former variants. To implement variational software, there exist various prominent techniques, such as preprocessors [5], feature-oriented programming [6], or clone-and-own [15]. Instead of maintaining every variant on its own, as it is done with clone-and-own, advanced techniques often use mechanisms to reuse code and variable code structures to generate the different variants [6, 16, 25]. As variational software is increasingly used for safety-critical systems [21], there is another concern that becomes more and more important: behavioral *correctness* of variational software.

*Correctness-by-construction* [14, 17, 19] is an approach to incrementally create correct programs. At first, the specification in form of pre- and postconditions is defined which is then refined into code using small, tractable *refinement rules*. Correctness-by-construction can be used supplementary to post-hoc verification [29], which verifies programs after their implementation in contrast to the incremental approach of correctness-by-construction. An advantage of using correctness-by-construction is that the code is not only created correctly according to the specification, but can also be simpler to understand, better structured, and more efficient than code that has been developed in an ad-hoc fashion into correctness [19].

CorC [24] is a tool that supports program development using correctness-by-construction. The program and its specification are written in Hoare triples consisting of a pre- and postcondition and a statement. These triples are translated into Java code and can typically be proven automatically [24] with the deductive verification tool KeY [4]. However, as the specifications with this approach are fixed, it is currently not possible to design variable code structures as required for variational software.

In this paper, we propose *variational correctness-by-construction* to combine the development of variational software with correctness-by-construction to efficiently construct software variants. Therefore, we introduce two key ingredients: (1) a mechanism to enable variability in statements and (2) a mechanism for variability in the specification. For the variability mechanism in statements, we use a keyword to mark the variation points in the method. These variation points can be resolved with different method calls to subsequently form distinct variants. As the behavior of the methods is changed via the variability mechanism in the statements, we adapt the specifications accordingly. To omit the fixed specifications that

are given when using correctness-by-construction, we use three different techniques of *contract composition* [27], such that contracts of different methods can be reused, adapted, or even overridden completely. To evaluate variational correctness-by-construction, we implement it in a tool called VarCorC[1] and used it to conduct two case studies including variants of certain methods. Thereby, we evaluate the feasibility and compare variational correctness-by-construction to post-hoc verification with JML contracts and KeY in terms of verification and specification costs. In summary, we make the following contributions:

- We propose variational correctness-by-construction as an extension of correctness-by-construction with support for variational software.
- We provide a prototypical implementation in an open-source tool called VarCorC.
- We evaluate variational correctness-by-construction in comparison to post-hoc verification with JML contracts and KeY in terms of verification and specification costs.
- We share two case studies that have been created using VarCorC as benchmark for further research.

## 2 CORRECTNESS-BY-CONSTRUCTION

*Correctness-by-construction* [19] is an approach to incrementally create correct programs guided by a specification. Every statement is surrounded by a specification in form of a pre- and postcondition, which form a *Hoare triple* of the form {P} S {Q}. Thereby, the precondition 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. P, Q, and the Hoare triple itself are *predicate formulas*, which means that they evaluate either to `true` or `false`. In this work, the pre- and postcondition are defined in *first-order logic* and the statement S is defined in *Guarded Command Language* [13], which uses the following five constructs: the empty command (*skip*), assignment (:=), composition (;), selection (**if**), and repetition (**do**). Additionally, we allow to use method calls in the statements.

The starting point for the development with correctness-by-construction is a Hoare triple with pre- and postconditions and an abstract statement. This triple can then be refined using *refinement rules* that guarantee the correctness at each refinement step. An abstract statement can also be replaced by a concrete program that satisfies the specification of the corresponding triple. The procedure is finished when no abstract statement is left. As every statement is surrounded by a specification, the program can be checked partially after each step. We present a list of the six most important refinement rules in Figure 1 and explain them in the following.

***Skip.*** A skip statement does not change the state of the program [13, 19].

***Assignment.*** An assignment statement sets an expression E of type T to a variable of the same type T. This rule replaces the abstract statement S in an Hoare triple with an assignment x := E if the precondition P implies the postcondition in which the variable x has been replaced by the expression E. Multiple assignments can be expressed in one statement as $x_0, x_1 := E_0, E_1$, where $E_0$ is the assignment for $x_0$ and $E_1$ is the assignment for $x_1$.

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

| {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 := E] |
| *3. Composition* : | {P} S1 ; S2 {Q} *iff there is an intermediate condition* M *such that* {P} S1 {M} *and* {M} S2 {Q} |
| *4. Selection* : | {P} **if** $G_1 \rightarrow S_1$ **elif** ... $G_n \rightarrow S_n$ **fi** {Q} *iff* (P *implies* $G_1 \vee G_2 \vee \ldots \vee G_n$) *and* {P ∧ $G_i$} $S_i$ {Q} *holds for all i.* |
| *5. Repetition* : | {P} **do** [I, V] G → S **od** {Q} *iff* (P *implies* I) *and* (I ∧ ¬G *implies* Q) *and* {I ∧ G} S {I} *and* {I ∧ G ∧ V=$V_0$} S {I ∧ 0≤V ∧ V<$V_0$} |
| *6. Method Call* : | {P} $M(a_1, ..., a_n, b)$ {Q} *with method* {P'} $M$(*parameter* $p_1, ..., p_n$, *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 [19]**

***Composition.*** The composition rule splits one abstract statement into two abstract statements $S_1$ and $S_2$. Additionally, an intermediate condition M has to be provided, which evaluates to `true` after $S_1$ and before $S_2$ is executed [13].

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

***Repetition.*** The repetition rule behaves similar to a while loop in other programming languages. As long as the guard G evaluates to true, the statement S is executed. 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.

***Method Call.*** The method call rule introduces a method with the following syntax: {P'} $M$(*parameter* $p_1, ..., p_n$, *return* $r$) {Q'} with $r$ representing a return variable of any type that gets assigned a new value after the execution of the method $M$, and $p_1, ..., p_n$ representing a list of parameters, whose scope is limited to the method body. Both, $p_1, ..., p_n$ and $r$ can also be empty, so that there are four combinations to define a method. In all cases, we omit side effects, as we define the parameters as call-by-value. Furthermore, the return value can not have side effects, as it is assigned to a variable that is treated by the conditions of the statement that calls the method (P and Q). Therefore, the method call may only have an impact on globally defined variables, which are covered by the pre- and postcondition of the calling statement as well. As a result, the method call refinement rule can generally be applied if the specification of the method complies with the specification of the statement. However, as the method $M$ uses the formal parameters $p_1, ..., p_n$ and $r$ in P' and Q' and the statement uses the

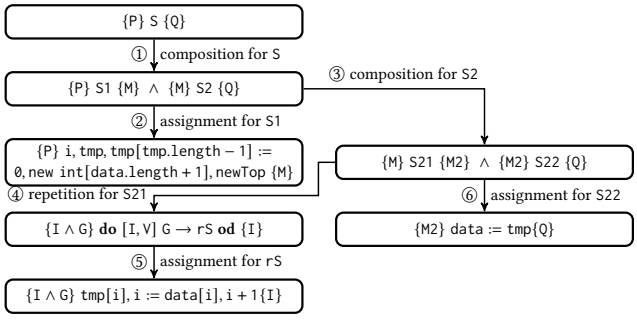


**Figure 2: Refinement Steps for the Method push (Adapted from [26])**

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'$[p_i \backslash a_i]$). As the scope of the formal parameters is limited to the method body, 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 $p^{old}$. To make the postconditions comparable, we therefore 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 be in the precondition, as it does not exist before executing the method.

## 3 MOTIVATING EXAMPLE

To showcase correctness-by-construction in practice, we exemplify it on the implementation of a push method of an *IntList* based on the implementation of Scholz et al. [26]. The *IntList* maintains an integer array data and offers the method push to add another element (newTop) to the array. In Section 3.1, we explain the programming style of correctness-by-construction using the refinement steps displayed in Figure 2 and show the finished algorithm of the method push in Guarded Command Language in Listing 1. Afterwards, in Section 3.2, we motivate variational correctness-by-construction by displaying the problems that occur if variants of the method push shall be developed efficiently.

### 3.1 Developing a Base Variant of Method Push

To start the development of the push method using correctness-by-construction, we first concretize the pre- and postcondition of the algorithm. As we have given an array with integers that shall contain at least one element, we set the precondition P to `data != null ∧ data.length > 0`. To simplify the postcondition, we use the predicates `contains(int[] A, int x)` and `containsAll(int[] A, int i, int j, int[] B)`. Thereby, the predicate `contains` evaluates to `true`, iff array `A` contains value `x` and `containsAll` evaluates to `true`, iff array `A` contains all elements of array `B` from index `i` (inclusive) to `j` (exclusive). In our example, the postcondition shall ensure that the array data contains the new element newTop and all elements that it has contained before the execution of the method push. The concrete postcondition using our self-defined predicates is defined as follows: `contains(data, newTop) ∧ containsAll(data, 0, data.length, data`$^{\text{old}}$`)`.

In our algorithm, we want to create a temporal array tmp, which shall contain all elements from data and additionally the new element newTop. Afterwards, we want to assign tmp to data to finalize the algorithm. In a first step, we apply the *composition refinement rule* ① to split the abstract statement S into two abstract statements. As we need to traverse through data later on in the algorithm, we already know that we need a loop counter variable. Additionally, tmp needs to be created with the length of data plus 1 and we can already add newTop as well. We can formalize these requirements in the intermediate condition M as follows: `tmp.length = data.length+1 ∧ i = 0 ∧ contains(tmp,newTop)`.

To fulfill this intermediate condition, we apply the *assignment refinement rule* ② to the first abstract statement S1. We define three assignments. First, we assign `0` to the loop counter variable `i`. Second, we create the array tmp with the length `data.length+1`. Third, we assign newTop to the last index of tmp.

For the second abstract statement S2 of the composition statement, we need to add every element of data to tmp and as a last step of the algorithm, assign tmp to data, as this is the data structure maintained by the *IntList*. Consequently, we need to split the second abstract statement S2 again using the *composition refinement rule* ③. As the assignment of tmp to data shall be the last step, we define the intermediate condition M2 as follows: `tmp.length = data.length+1 ∧ contains(tmp,newTop) ∧ containsAll(tmp,0,data.length,data)`.
The array tmp shall contain newTop and all elements from data, as these are the conditions that need to be fulfilled for data after the assignment as well.

As a next step, we further refine the first abstract statement of the second composition (S21). The parts of the intermediate condition M2 `contains(tmp,newTop)` and `tmp.length = data.length+1` are already fulfilled by tmp. Therefore, we only need to fulfill the remaining part of M2, which is `containsAll(tmp,0,data.length, data)`. This can be achieved by applying the *repetition refinement rule* ④. It requires the additional definition of a guard, an invariant, and a variant to guarantee the correct postcondition and the termination of the loop. We select `i < data.length` as the guard, as we want to traverse data from left to right. The variant can be chosen as `data.length - i` using the loop counter variable `i`, which is monotonically increasing in the loop. Finally, we define an invariant for the loop. We use the conditions `tmp.length = data.length+1 ∧ contains(tmp,newTop)` from the intermediate condition M and add the condition `containsAll(tmp,0,i,data)`, which reflects the behavior of adding each element from data to tmp during the repetition of the loop. To implement the loop body, we refine the abstract repetition statement rS using the *assignment refinement rule* ⑤. We assign the value of data at index `i` to tmp at the same index and increase `i` by 1.

To finalize the algorithm, we need to concretize the last abstract statement S22 by assigning tmp to data using again the *assignment refinement rule* ⑥.

### 3.2 Variants of Push

In Section 3.1, we developed a method to add an additional element to an integer array. Now, we want to create a variant of this algorithm, which shall keep the array sorted. Sticking with correctness-by-construction, we would have to repeat the same

```
1 pre: data ≠ null & data.length > 0
2 post: contains(data,newTop) & containsAll(data,0,data.
      length,data^old)
3 i,tmp,tmp[tmp.length-1] := 0,new int[data.length+1]
      ,newTop;
4 do i < data.length →
5     tmp[i],i := data[i],i + 1;
6 od
7 data := tmp
```

**Listing 1: Method push in Guarded Command Language**

```
1 pre: data ≠ null & data.length > 0
2 post: contains(data,newTop) & containsAll(data,0,data.
      length,data^old) & isSorted(data)
3 i,tmp,tmp[tmp.length-1] := 0,new int[data.length+1]
      ,newTop;
4 do i < data.length →
5     tmp[i],i := data[i],i + 1;
6 od
7 data := tmp;
8 sort(data,data)
```

**Listing 2: Sorted Variant of Method push in Guarded Command Language**

steps as described in the previous section starting with a changed postcondition, as we need to express the new behavior of our sorted push method. The complete algorithm in Guarded Command Language is shown in Listing 2. Comparing it to the base variant in Listing 1, we can see that the general structure stayed the same. In fact, regarding the implementation only an additional method call is added in Line 8, so that more duplicated code than actually changed code has been created.

As a result, if we need to change one of these cloned lines in one of the variants, the change probably needs to be applied to the other variant as well. The more variants are created this way, the higher the maintenance costs become. To reduce these costs, we want to use a mechanism that allows to reuse code from former variants. Therefore, we use the keyword `original` to mark the variation points in the code, which can later be replaced to form distinct variants. In our example, we would implement the sorted refinement as seen in Listing 3. We added the variation point in Line 3, which can be replaced by the base variant in Listing 1 to create the variant of the sorted push method. However, the postcondition of the base variant is not sufficient anymore, as we refined the method and added functionality. The specification as described in Listing 2 would work in this case. However, we would create specification clones from the conditions of the base variant, which causes higher maintenance costs again.

An additional issue arises when we want to refine the method more than once. For instance, one could think of a variety of different refinements for the push method like a refinement that adds a second element, a refinement that limits the amount of elements in the array, or a refinement that ensures that the array does not contain duplicated values. All of these refinements require slightly different specifications and they can be used optionally in a refinement chain to compose a distinct variant of the push method. Consequently, when generating a distinct variant using three or four of these refinements we need to refer to the specifications of the single refinements without knowing which exact ones have been selected.

```
1 pre: ???
2 post: ???
3 original(data,newTop);
4 sort(data,data)
```

**Listing 3: Variation Point for the Sorted push Method**

# 4 CORRECTNESS-BY-CONSTRUCTION FOR VARIATIONAL SOFTWARE

As illustrated in the previous section, correctness-by-construction cannot be applied directly to variational software without constructing code clones which involves redundant effort. Therefore, we use two building blocks to extend correctness-by-construction to variational correctness-by-construction. The first building block extends the refinement rules to enable variability in the statements. The second building block comprises a mechanism to enable variability in the pre- and postcondition of the starting triple. Both building blocks are explained in the following.

## 4.1 Variation Points

The first building block to construct variational software with correctness-by-construction is to add an additional refinement rule that allows to mark variation points in a method as demonstrated in Listing 3. The mechanism we use for this part is inspired by feature-oriented programming [6]. Thereby, the programmer creates different refinements of a method that override each other in a distinct order. However, these method refinements can use the keyword `original` to call the implementation of one of the other refinements of this method. The order in which the different refinements of that method are composed can be varied to form the desired variant of that method. Basically, an original call behaves similar to a super call in Java. However, as we create a whole hierarchy and every refinement apart from the base method may call `original`, the contracts of the replacing method needs to be composed as well. We define the variation point refinement rule as follows:

$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to*
$\{\mathtt{P}\}$ **original**$(a_1, ..., a_n,\ b)$ $\{\mathtt{Q}\}$ *with m composed methods* $R =$
$\{\mathtt{P'}\}\ M(param\ p_1, .., p_n,\ return\ r)\ \{\mathtt{Q'}\}$ *which are composed as*
$c_1 \bullet c_2 \bullet \ldots \bullet c_x$ *with x method refinements* $c_i =$
$\{\mathtt{P}_i\}\ M(param\ p_1, ..., p_n,\ return\ r)\ \{\mathtt{Q}_i\}$
*iff for all R:* $\mathtt{P}$ *implies* $\mathtt{P'}[p_j \backslash a_j]$ *and* $\mathtt{Q'}[p_j^{old} \backslash a_j^{old}, r \backslash b]$ *implies* $\mathtt{Q}$

As keyword `original` can be replaced by several different composed methods the variation point refinement rule is based on the method call refinement rule we introduced earlier in Section 2. Therefore, the implications in the last row are identical to the implications in the method call refinement rule. However, as the methods can be refined multiple times there are also multiple valid replacements for the `original`. To ensure correctness we consequently have to guarantee that all valid replacements `R` comply with the calling statement. Thereby, a valid replacement has to provide a contract in form of a Hoare triple and the same signature as the calling method. The contract of the replacing method can be the result of the composition of multiple method refinements (i.e., a refinement chain of the form $c_1 \bullet c_2 \bullet \ldots \bullet c_x$). The method refinements also have to offer a contract and be defined with the same signature. Apart from these restrictions, the user can define the

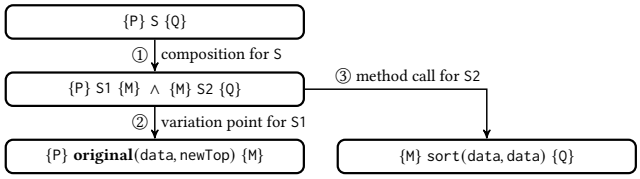


**Figure 3: Refinement Steps for the Sorted Refinement of push**

valid method replacements and their refinement chains themself. The composition of the method refinements in the refinement chain for a replacing method is performed using contract composition, which comprises several different techniques to compose pre- and postconditions. We describe the techniques in the following subsection.

*Example 4.1.* In Figure 3, we show the refinement steps using the variation point refinement rule for the sorted refinement in the context of the method push we introduced earlier in Section 3. We only defined this refinement and a base variant, therefore the only replacement of the variation point is the base variant and as it is a single method and no refinement chain, we do not need to compose the contracts to form `P'` and `Q'`. In other words, the conditions of the base variant as defined in Listing 1 directly correspond to `P'` and `Q'` from the variation point refinement rule.

## 4.2 Contract Composition

In this subsection, we introduce our mechanism for variability in the specification of programs written with correctness-by-construction. We apply contract composition in two places: First, to compose pre- and postconditions of the whole method (cf. Section 3.2) and second, to compose the contract of the method call that replaces the variation point (cf. Section 4.1).

Correctness-by-construction uses contracts in form of Hoare triples with pre- and postconditions that reflect the behavior of the code to verify it. As variational software modifies methods and may also completely change a method's behavior in a way that violates the Liskov principle [27], it is crucial to be able to compose their contracts as well. For example, when a new variant adds functionality to a method by refining it, the old contract of that method will most likely be insufficient to verify its behavior (cf. Section 3.2). In that case, the contract needs to be adapted as well using a specific composition technique.

Some of the authors extended the composition of feature modules in feature-oriented programming by the contracts and proposed six composition techniques [27]. Based on their empirical evaluation, we chose three of them (*explicit contracting*, *contract overriding*, and *conjunctive contract refinement*), as they have been shown sufficient for the case studies that are used in our evaluation. We applied these techniques to the pre- and postcondition.

To introduce the mechanisms formally, we refer to a contract $c$ in the form of $c = \{\phi\}m\{\psi\}$, with $\phi$ being the precondition, $\psi$ the postcondition, and $m$ the method. We define the original contract $c$ with $c = \{\phi\}m\{\psi\}$, the refining contract $c'$ with $c' = \{\phi'\}m'\{\psi'\}$, and denote the composed contract as $c'' = c' \bullet c = \{\phi'\}m'\{\psi'\} \bullet \{\phi\}m\{\psi\} = \{\phi''\}m' \bullet m\{\psi''\}$. The specific mechanisms for the contract composition defines how $\phi''$ and $\psi''$ are derived from the contracts $c$ and $c'$. We consider a contract composition mechanism M as a function $\bullet_M : C \times C \rightarrow C$ defined over the set $C$ of all possible contracts. We require that each contract $c \in C$ can be formulated as $c = \{\phi\}m\{\psi\}$ regardless of the particular specification language.

*Conjunctive Contract Refinement.* This mechanism composes two contracts implicitly, without making the composition visible to the contract [27]. In particular, conjunctive contract refinement conjuncts the pre- and postcondition of the original and refined contract to form the composed contract. Respectively, we define the conjunctive contract composition as

$$c \bullet_{\text{ccr}} c' = \{\phi' \wedge \phi\}m' \bullet m\{\psi' \wedge \psi\}$$

*Example 4.2.* To showcase the application of the composition techniques, we apply them on the pre- and postcondition of the sorted refinement of the method push from the *IntList* that we introduced earlier in Section 3. For conjunctive contract refinement, we can define the pre- and postcondition in Listing 3 as `P = true` and `Q = isSorted(data)`. When composing the base variant in Listing 1 with the sorted refinement using conjunctive contract refinement the composed pre- and postcondition would result in `P = data ≠ null ∧ data.length > 0 ∧ true` and `Q = contains(data, newTop) ∧ containsAll(data, 0, data.length, data`$^{\text{old}}$`) ∧ isSorted(data)`.

Conjunctive contract refinement offers the possibility to reuse original contracts without having extra specification effort and therefore reduces specification clones, because original contracts are assumed to hold in all cases. Additionally, the composed contracts are easy to understand because the conditions of the different method refinements are simply combined with a conjunction. However, the main disadvantage of this composition technique is that all method refinements have to be known in advance, as they need to be fulfilled in every case. This prohibits the modular reasoning and makes it hard to understand and maintain the methods. Another disadvantage is the reduced flexibility of the mechanism. Contracts can only be refined by adding formulas via conjunction to the existing pre- and postconditions.

*Explicit Contracting and Contract Overriding. Explicit Contracting* offers the possibility to use a specific keyword to reference the original pre- or postcondition in the refining contract [27]. We use the keyword `original` and define explicit contracting as follows:

$$\{\phi'\}m'\{\psi'\} \bullet_{\text{ecr}} \{\phi\}m\{\psi\} = \{\phi' \bullet \phi\}m' \bullet m\{\psi' \bullet \psi\}$$

In this context, $\{\phi' \bullet \phi\}$ is defined as replacing every occurrence of `original` in $\phi'$ by $\phi$, whereas $\{\psi' \bullet \psi\}$ is defined analogously. However, the occurrence of the keyword `original` is not mandatory and it can even be used multiple times.

In fact, *contract overriding* is a special form of explicit contracting when no `original` is used in the conditions. In that case, the original contract is simply overridden by the refining contract. Nevertheless, we decided to integrate both techniques to force the programmer to choose the technique deliberately. Contract overriding can be formalized as:

$$c' \bullet_{\text{co}} c = c'$$

*Example 4.3.* With contract overriding, we can define the specification in Listing 3 as `P = data ≠ null ∧ data.length >`

`0` and `Q = contains(data, newTop)` ∧ `containsAll(data, 0, data.length, data`$^{\texttt{old}}$`)` ∧ `isSorted(data)` for the precondition `P` and the postcondition `Q` postcondition. Comparing these to the conditions from the base variant in Listing 3, we can see that parts of the postcondition and the whole precondition have been cloned.

When applying explicit contracting to the same example, we can now reference the pre- and postcondition of the base variant using the keyword `original` to avoid specification clones. We can define the precondition as `P = original` and the postcondition as `Q = original` ∧ `isSorted(data)`. The `original` would be resolved as the conditions from Listing 1 when composing the base variant with the sorted refinement of the push method. The composed contract would be equal to the one we defined with contract overriding.

Explicit contracting is an intuitive approach to compose contracts, as it offers the same linguistic means as we use for variation points (cf. Section 3.2). The programmer can freely decide if, where, and how often to use the keyword `original` to reference the original contract. Therefore, the pre- and postconditions can independently be overridden, refined, or even reused completely. By offering the ability to refer to the original contract, this mechanism overcomes the drawback of having many specification clones, as we have seen for contract overriding (cf. Example 4.3). However, the caller needs to be aware of any contract that has been defined for a specific method. In the example of the push method, the composed postcondition is sufficient for the variant of the sorted array, but it may not if another refinement is selected. Additionally, the pre- and postconditions can be refined independently and be negated or included in further logical constructs. Another drawback of referencing the original contracts is the possibility of dangling references. This occurs, when composing contracts in a distinct order, but when the `original` is called, there is no original contract defined for this specific configuration. In Example 4.3, this could occur, if we want to create a variant using only the refinement in Listing 3 without the base variant in Listing 1. When composing the contract for this variant, the keyword `original` has to be replaced by a non-existent condition, which leads to the error.

*Refinement Chains.* For simplicity, we always used a base variant and one refinement for it to demonstrate the functionality of the previously introduced contract composition techniques (cf. Examples 4.2 and 4.3). However, one method can also be refined multiple times by building a *refinement chain*. Thereby, a refinement chain consists of $n$ different refinements that can be composed to form a distinct variant of the method ($c_1 \bullet c_2 \bullet \ldots \bullet c_n$) and the composition technique can be chosen independently for each refinement.

## 5 TOOL SUPPORT: VARCORC

VarCorC[2] is an open-source Eclipse plugin that implements variational correctness-by-construction. It is based on CorC [24], which already supports programming with correctness-by-construction. At its core, it has a correctness-by-construction meta-model modeled with the Eclipse Modeling Framework.[3] It offers a textual and a graphical editor which both use the underlying meta-model. The textual editor is implemented with XText[4] and the graphical one with Graphiti.[5] To prove the correctness of the refined Hoare triples the deductive verification tool KeY [4] is used.

KeY is usually used for post-hoc verification, where formal methods and tools are applied on the finished program to prove that the program satisfies the specification. The specification is typically added to the program as annotations in the form of pre- and postconditions to every method in a class. Post-hoc verification tools, like KeY, use a formal calculus to prove the program correct with respect to its specification semi-automatically. Therefore, VarCorC generates a problem file that formulates the distinct refinement rules and conditions so that KeY can read and try to verify them. In case of the variation point rule, the user has to additionally define all refinement chains that can possibly replace the variation point so that the distinct variants can be verified by KeY. Thereby, VarCorC generates a problem file for each refinement chain composing (1) the refinements and (2) the pre- and postconditions according to the specified composition techniques (cf. Section 4.2).

The graphical editor of VarCorC offers a tree structure starting with one Hoare triple at the top that can be refined as desired by the user. The resulting structure looks similar as seen in Figure 2. Thereby, the pre- and postcondition from the top are propagated automatically to the refinements. Additionally, the user can define variables and global conditions. The variables have a name, a type, and kind (parameter, return value, or local) to define the signature of the method that is implemented in the diagram. The global conditions that have been defined have to be fulfilled by all refinement steps. The graphical editor also encompasses a visualization indicating whether the program has been successfully proven by KeY. The verification process is triggered by right-clicking any statement in the tree and choosing the verify-command in the context menu. Thereby, the subtrees can be verified independently.

## 6 CASE STUDY

To evaluate feasibility of variational correctness-by-construction, we applied it to two prominent product lines, namely *IntList* and *BankAccount*. Both were already used (1) for *specifying* software product lines [27] and (2) as case studies for CorC [24]. In the following, we describe the settings of our case study and present and discuss results.

### 6.1 Settings

In particular, we address the following three research questions:

- **RQ1:** Is it possible to develop variational software using variational correctness-by-construction?
- **RQ2:** What are the verification costs compared to post-hoc verification?
- **RQ3:** What are the specification costs compared to post-hoc verification?

**RQ1** gives us insights to what extent variational correctness-by-construction can be used to create correct variational software. By answering **RQ2**, we may estimate whether variational correctness-by-construction reduces the verification costs, and therefore, may be even more attractive than post-hoc verification in certain situations.

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

| Subject | Method | Refinements | Variants |
|---|---|---|---|
| *IntList* [26] | push | 3 | 4 |
| *BankAccount* [28] | update | 2 | 2 |
| | undoUpdate | 2 | 2 |
| | nextDay | 3 | 4 |
| | nextYear | 2 | 2 |

**Table 1: Subjects and their Characteristics**

By comparing the specification costs in **RQ3**, we can assess the relationship between verification and specification costs.

To answer the research questions, we implemented all variational methods of the two mentioned case studies *IntList* and *BankAccount*. Both case studies already exist as Java programs specified with JML, and were therefore transformed into VarCorC programs. *IntList* resembles a single class `IntList` and consists of methods to add or remove elements from an integer array. Here, method `push`, which inserts elements into the list, is implemented with three refinements. *BankAccount* provides its core functionality in class `Account`, which comprises a total of four variational methods to manage a bank account. Methods `update` and `undoUpdate` manipulate the savings of the account, and `nextDay` and `nextYear` allow to reason about withdrawal limits of users. In Table 1, we summarize the five methods including their characteristics. To enable the comparison to post-hoc verification, each variant of a method is implemented as a CorC program and a Java program annotated with JML. To answer **RQ2** and **RQ3**, we verify each variant and compare the verification costs regarding total proof nodes that are needed to close the proof. The specification cost is measured by counting the amount of conditions that have been connected using a conjunction in all user provided specifications. This includes the pre- and postconditions, class invariants, and additionally the intermediate conditions, which only occur in CorC. Loop invariants are identical in both cases and are therefore omitted.

## 6.2 Results and Discussion

We divide this section according to our three research questions.

***RQ1: Is it possible to develop variational software using correctness-by-construction?*** We manually created twelve programs in VarCorC, one for each method refinement of the two case studies (cf. Table 1). More precisely, VarCorC supports variation points similar to feature-oriented programming by using keyword `original` (cf. Section 4.1) and the composition of specifications using one of the three discussed composition techniques (cf. Section 4.2). Both concepts allowed us to exactly rebuild the behavior as intended in other studies [27]. In total, 14 different methods could be derived with respect to possible feature combinations. As expected, all proofs passed for all 14 variants, which means that guarantees were preserved during feature composition. We conclude that, with VarCorC, it is possible developing correct variational software using variational correctness-by-construction.

***RQ2: What are the verification costs compared to post-hoc verification?*** We collected the proof nodes as an indicator for the verification effort for every verified variant for both approaches. The accumulated results for each method are shown in Figure 4.

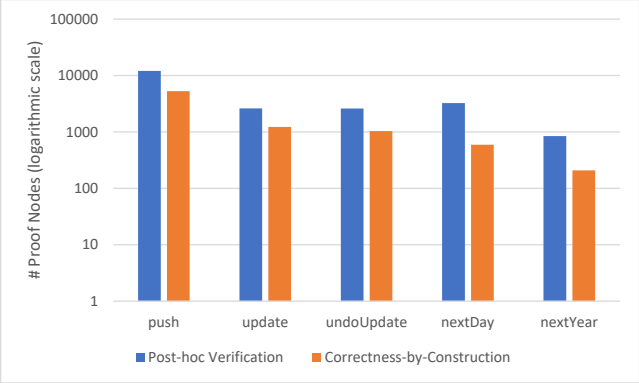


**Figure 4: Total Amount of Proof Nodes per Method**

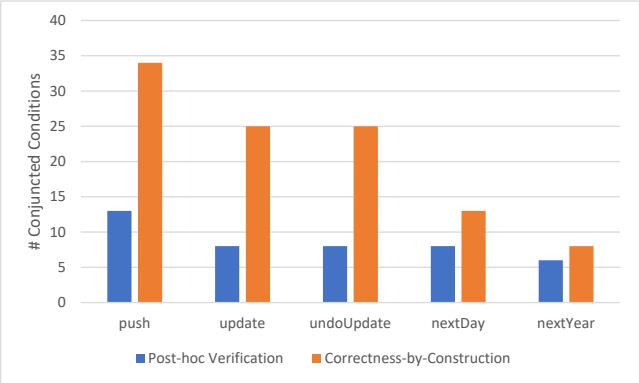


**Figure 5: Total Amount of Conjuncted Conditions per Method**

The scaling of the y-axis is logarithmic. The verification with VarCorC needs between 53% and 81% less nodes than the verification with Java and JML. This result is in alignment with our expectations and related work [24] by indicating a reduced proof complexity, as the verification of one method is split into smaller problems using correctness-by-construction. This means that even for smaller case studies, the total number of proof steps needed for VarCorC is at least twice as small as the number of proof steps needed for post-hoc verification. The effects may be considerably larger with more complex and varied case studies.

***RQ3: What are the specification costs compared to post-hoc verification?*** In Figure 5, we illustrate the total specification costs for VarCorC and JML per method refinement with respect to precondition, postcondition, intermediate condition (only VarCorC), global conditions (only VarCorC), and class invariants (only JML). We found that the average amount of conjuncted conditions used for VarCorC were 62% higher for *IntList* and 58% for *BankAccount*. However, these metrics do not fully reflect the complexity of the single conditions as the conjuncted conditions itself range from rather complex (e.g., using an `\exists` clause) to quite simple (e.g., assuming the value of a variable to be zero). Nevertheless, even simple conditions, such as an object to be non-null, mean effort as they still have to be specified, even if manual effort is lower compared to more complex conditions. We also found that 58% of the extra

conditions have been introduced by intermediate conditions, which tend to be more complex than the global conditions. However, our experience was that also most of the intermediate conditions did not take too much effort to specify, as correctness-by-construction is applied on the fine-grained level of statements. Furthermore, when the postcondition is already known, the intermediate conditions often slowly build up and partially reflect the postcondition. Still some intermediate conditions have been more complex (e.g., if the statement contained an `original` that could be replaced by more than one method). In summary, there is a trade-off in VarCorC between specification and verification costs.

## 7 RELATED WORK

In this paper, we mainly addressed correctness-by-construction for the verification and implementation of variational software. Therefore, we will present related verification and implementation techniques for variational software and other refinement-based approaches similar to correctness-by-construction in the following.

*Feature-oriented programming* [6, 7] is a technique to implement software product lines, which is a family of related software products that share a common code base and are each composed by a valid combination of software artifacts [12]. The variability mechanism of feature-oriented programming is related to the functioning of our variability mechanism. However, we formalized the mechanism and defined rules to guarantee its correctness for variational correctness-by-construction. Another technique to develop software product lines that is similar to feature-oriented programming is called *delta-oriented programming*. Basically, delta modules compass the same functionality as feature modules, but additionally they also allow to remove classes and members [25]. Neither feature-oriented programming nor delta-oriented programming compass specifications to develop correct programs which is the main difference to variational correctness-by-construction.

Some of the authors defined six *composition techniques* for feature-oriented contracts and applied them to software product lines implemented with Java and specified with JML contractss [27]. For variational correctness-by-construction, we adapted three of these composition techniques to correctness-by-construction to compose the conditions of the different statements and to form the contracts for the original calls. Bruns et al. [8] 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 [18] propose another composition technique for delta-oriented programming, which is implemented as a restrictive form of the explicit contracting. However, all of these approaches apply their mechanism to design-by-contract contracts based on JML and use post-hoc verification for the proofs, which is a difference to the incremental approach of stepwise refinement that we apply by using variational correctness-by-construction.

In the following, we will present related work for *correctness-by-construction*. The Event-B framework [2] uses automata-based systems including a specification 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. Thereby, 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 also work on automata-based systems rather than on code and specifications.

ArcAngel [22] 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 [23] 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 [11] extends CRefine by adding refinement tactics.

The tools iContract [20] and OpenJML [10] both apply design-by-contract to Java code by using a special comment tag to insert conditions. These conditions are translated into assertions and checked at runtime which is the main difference to VarCorC because they do not use formal verification.

## 8 CONCLUSION

*Variational software* has become common practice to manage the growing demand for software variants in one specific domain. As it is increasingly used in safety-critical systems, the verification of these systems becomes more important, which is a challenge due to the increasing amount of variants.

In this paper, we proposed our methodology called *variational correctness-by-construction* for the development and verification of variational software. At first, we defined the variation point refinement rule as an extension to correctness-by-construction to introduce variability in the implementation. Second, we presented contract composition for refined method contracts to allow variability in the pre- and postcondition. Third, we implemented variational correctness-by-construction in a tool called VarCorC to evaluate our concepts in terms of feasibility and specification and verification costs compared to post-hoc verification with JML contracts and KeY.

Our main insight is that our work has just been a starting point on this subject. However, we showed feasibility on five variational methods and were able to verify them with significantly less nodes compared to post-hoc verification with KeY. These results show the reduced proof complexity and also the potential of variational correctness-by-construction. The high specification costs alleviate the reduced verification costs, but we argue that they can be further reduced by extending the tool support.

For future work, we plan on proving soundness of the proposed verification point refinement rule and further evaluate variational correctness-by-construction to strengthen our evaluation results. Therefore, one could use more case studies that have been retrieved in different ways also including larger ones with methods that have been refined multiple times. Another possiblity is to expand variational correctness-by-construction to the context of software product lines using a composition-based approach and feature models to model the variability in terms of features. In a next step, one could then work out a family-based verification and compare its efficiency to the current technique.

# 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] Michalis Anastasopoules and Critina Gacek. 2001. Implementing Product Line Variabilities. 26, 3 (2001), 109–117.

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

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

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

[9] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry. 91 (2014), 3–23.

[10] David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. 472–479.

[11] 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.

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

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

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

[15] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. 25–34.

[16] 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.

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

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

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

[20] R. Kramer. 1998. iContract - The Java(Tm) Design by Contract(Tm) Tool. 295–307.

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

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

[23] 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.

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

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

[26] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. 2011. Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report. Article 7, 7:1–7:8 pages.

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

[28] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. 2014. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. 177–186.

[29] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. 2016. Correctness-by-Construction and Post-Hoc Verification: A Marriage of Convenience?. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 730–748.