Re-CorC-ing KeY: Correct-by-Construction Software Development based on KeY

Tabea Bordis[1,2], Loek Cleophas[3,4], Alexander Kittelmann[1,2], Tobias Runge[1,2], Ina Schaefer[1,2], and Bruce W. Watson[4,5]

[1] Karlsruhe Institute of Technology (KIT), Germany
[2] TU Braunschweig, Germany
[3] TU Eindhoven, The Netherlands
[4] Stellenbosch University, South Africa
[5] School for Data-Science & Computational Thinking
Stellenbosch University, South Africa
{tabea.bordis,alexander.kittelmann,tobias.runge,ina.schaefer}@kit.edu,
l.g.w.a.cleophas@tue.nl, bwwatson@sun.ac.za

Abstract. Deductive program verification is a post-hoc quality assurance technique following the design-by-contract paradigm where correctness of the program is proven only after it was written. Contrary, correctness-by-construction (CbC) is an incremental program construction technique. Starting with the functional specification, the program's correctness is guaranteed by application of a small set of refinement rules. Even though CbC is supposed to lead to code with a low defect rate and improve the traceability of errors, it is not widespread. One of the main reasons is insufficient tool support which we addressed with our tool CorC. CorC provides support for CbC-based program construction with the KeY program verifier as backend prover for checking correctness of refinement rule applications. However, CorC was limited to constructing single method bodies restricting its applicability. In this work, we introduce and discuss CorC 2.0, which extends CorC's programming model with objects as used in object-oriented programming. We integrate CorC into a development process that allows to use post-hoc verification and CbC interchangeably to construct correct programs, and scale the applicability of CbC on the architectural level in our tool extension ArchiCorC. We developed three object-oriented case studies and evaluated the verification effort and the usability of CorC in comparison to post-hoc verification.

1 Introduction

The amount of software in safety-critical systems increases, and, therefore, functional correctness of programs is an important concern. While most verification approaches rely on post-hoc verification [13, 23, 50, 51], where a program is only verified after it is implemented, the stepwise *correctness-by-construction* development approach (CbC) as proposed by Dijkstra [20], Gries [21], or Kourie and Watson [29] offers an alternative approach. A behavioral specification in form

of a pre- and postcondition pair is refined into code using a set of tractable *refinement rules*. To guarantee the correctness of the refinement steps, each rule defines specific side conditions for its applicability. As a result, when applying CbC compared to classical post-hoc verification, errors are more likely to be detected earlier in the design process [35].

Our long-term vision is to make CbC accessible for large-scale software development. CorC [43] is a tool based on the deductive program verifier KeY [3] that supports the development of single methods following the CbC paradigm as imagined by Dijkstra [20]. The program and its specification are separated into several Hoare triples, each triple consisting of a pre- and postcondition pair and a statement written in Java. These triples can typically be proven automatically [43] with KeY. Thus, CorC adds tool support for CbC-based development to the KeY ecosystem. So far CorC could only be used to develop single algorithms as methods, independent of classes or larger software systems. One major stepping stone towards our vision to extend the applicability of CbC-based development is a development process that uses post-hoc verification and CbC in concert to take advantage of both approaches. In this paper, we focus on integrating object-orientation and a roundtrip engineering approach into the new version of CorC, called CorC 2.0. CorC 2.0 covers the same object-orientation language concepts that KeY covers for post-hoc verification and enables the combination of classic post-hoc verification using KeY and CbC-based development to improve the development of correct Java programs.

We extend CorC by four features such that object-oriented Java programs can be created using CbC and integrated into existing Java projects.

Graphical View. We provide a graphical view to create classes with fields, class invariants, and methods.

Inheritance and Interfaces. We support constructive development with interfaces and inheritance using the Liskov principle.

Roundtrip Engineering. We implement a roundtrip engineering approach such that existing Java classes can easily be imported into CorC to show their correctness and afterwards exported back to the original project as verified Java code. Thereby, the developer can freely decide which parts of the software shall be constructed using CbC in CorC and which parts shall be verified with post-hoc verification (or even stay unverified if it is not a safety-critical part).

Change Tracking. We use a change tracking mechanism that simplifies ongoing development by marking the refinement steps that have to be re-verified due to changing contracts and implementations of methods or classes (e.g. method calls, inherited method contracts, changed interface specifications).

To evaluate our concept, we recreate three case studies containing multiple classes and methods and compare the verification effort in terms of needed proof steps and verification time to deductive verification with KeY. By doing so, we found that with CorC 2.0 we were able to prove a larger number of methods compared to post-hoc verification with KeY, and that CorC 2.0 also sometimes

outperforms post-hoc verification with respect to verification effort. To further extend CbC-based development, we give an outlook on CbC-based development of component-based systems on the architectural level in our tool ArchiCorC and on CbC-based development of feature-oriented software product lines in our tool VarCorC.

2 Related Work

Besides of the CbC-based program construction approach proposed by Dijkstra [20] and Kourie and Watson [29], which we pursue in this work, there are other *refinement-based approaches* that guarantee the correctness of the program under development. In the Event-B framework [1], automata-based system descriptions are refined to concrete implementations. This approach is implemented in the Rodin platform [2]. In comparison to the CbC approach used here, the abstraction level is different. CorC uses specified source code instead of automata as main artifact. Morgan [36] and Back [8] also proposed related CbC approaches. Morgan's refinement calculus, which comprises a very large number of different refinement rules in comparison to the minimal set of refinement rules in CorC, is implemented in the tool ArcAngel [37]. Back et al. [6, 7] developed the tool SOCOS. In comparison to CbC in CorC, SOCOS works with invariants in contrast to pre-/postcondition specifications as used in CorC.

Another field that emerged from the ideas of Gries [21] and Dijkstra [20] is *program synthesis*. Program synthesis is the task to *automatically* find a program that satisfies a formal specification provided by a developer. Foundational work has been proposed by Manna and Waldinger [33] and has been continued by many others. For example, Stickel et al. [48] propose a deductive approach that extracts a Fortran program from a user-given graphical specification by composing entries of subroutine libraries. Gulwani et al. [22] propose a component-based synthesis that generates and resolves so called synthesis constraints and apply their approach to bitvector programs. Heisel [25] uses a proof system that builds up a proof during program development. Polikarpova et al. [41] propose an approach to synthesize recursive programs from a specification in the form of a polymorphic refinement type. In contrast to program synthesis, CbC as we apply it does not automatically synthesize code from a specification. CbC is rather a development approach that is guided by a specification and guarantees correctness by proving that the side conditions that are introduced by the set of refinement rules are fulfilled. The developer therefore still has control over the program resulting from the approach while with program synthesis one of possibly many implementations that fulfill the specification is generated. Furthermore, program synthesis has limitations regarding scalability, as for example recursive programs including loops are hard to synthesize.

Some of the authors' recent work on trait-based CbC [44] proposes to replace meta-refinement rules of CbC that are outside of the programming language with trait-based program development and trait-based composition. Refinement in TraitCbC amounts to implementing an abstract method in a trait by a con-

crete method in another trait and composing those two traits to realize the abstract method by the concrete implementation. TraitCbC per se is parametric in the specification language, meaning that any trait-based language with a corresponding specification language and verification framework can be used to instantiate TraitCbC. In some of the authors' implementation [44], we use KeY [3] to establish the correctness of traits and trait composition. Abstract execution [47] in KeY allows verifying the correctness of methods with abstract program parts, which are specific by contracts. Abstract execution can also be used for refinement-based CbC where abstract program parts are incrementally refined to more concrete program parts. This allows for a fine-grained adaption of the granularity of refinement that ranges between single program statements (as in CorC) and whole methods (as in TraitCbC). The main difference in abstract execution is however that it extends the programming language with abstract program parts and, thus, allows to better reason about irregular termination (e.g., break/continue) of methods.

KeY [3] is a deductive program verifier for Java programs specified with the Java modeling language. KeY is the verification backend of CorC. Besides KeY, there are a number of tools for program specification and verification, such as: the language Eiffel [34] with the verifier AutoProof [53], the languages SPARK [9], Dafny [31], and Whiley [38], and the tools OpenJML [15], Frama-C [17], VCC [14], VeriFast [26], and VerCors [4]. All those are candidates for post-hoc verification tools to be compared with the CbC methodology, or they can serve as backends for guaranteeing the correctness of rule applications in refinement-based CbC, similar to the usage of KeY in CorC. In CorC, we decided to use KeY as backend because of previous familiarity and support from the KeY developer and user community.

3 Correctness-by-Construction in CorC

In this section, we introduce correct-by-construction software development in the previous version of CorC. In Section 3.1, correctness-by-construction in CorC is introduced by an example. In Section 3.2, we present the basic functionality of CorC with the graphical and textual editor.

3.1 Correctness-by-Construction

The correctness-by-construction program development approach starts with a Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$. An abstract program S is refined stepwise into a correct implementation by applying refinement rules. A refinement rule concretizes the Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$ to $\{\mathsf{P}'\}\ \mathsf{S}'\ \{\mathsf{Q}'\}$. For example, a loop, a conditional statement, or a sequence of statements could be introduced as S'. CbC by Kourie and Watson [29] offers a set of refinement rules that guarantee the correctness of the refined program if specific side conditions hold.

In Figure 1, we show an example of a linear search algorithm that is created with the graphical CorC tool. Each node with a green border represents a

Hoare triple and corresponds to the application of one refinement rule. We start with the program $\{\mathsf{P}\}\mathtt{statement}\{\mathsf{Q}\}$ at the top, where `statement` is an abstract statement. The starting precondition $\mathsf{P} := \mathtt{appears}(\mathtt{a}, \mathtt{x}, 0, \mathtt{a.length})$ states that the element `x` appears in the array `a` inside the boundaries of the array. Here, `appears` is a predicate to shorten the specification. The postcondition $\mathsf{Q} := \mathtt{modifiable}(\mathtt{i});\ \mathtt{a}[\mathtt{i}] = \mathtt{x}$ states that the element `x` is in the array at position `i`. We specify with the predicate `modifiable` that only `i` can be altered in the program. In CorC, the accessible variables are defined in a variables box, which is shown on the right side. The function has two parameters `a` and `x` and a local variable `i`. The global conditions box specifies conditions which are valid in every step of the program (i.e. invariants), and hence added implicitly to every Hoare triple.

The first refinement is the application of the composition rule [29], which splits the starting Hoare triple by triples $\{\mathsf{M}\}\mathtt{statement1}\{\mathsf{M}\}$ and $\{\mathsf{M}\}\mathtt{statement2}\{\mathsf{Q}\}$ with an intermediate condition M. The idea of the algorithm is that we traverse the array from back to front and stop, when the element `x` is found. The invariant of the program is $!\mathtt{appears}(\mathtt{a}, \mathtt{x}, \mathtt{i}+1, \mathtt{a.length})$. If we have not yet stopped, we know that the element `x` does not appear in the end of the array that is already examined. We also use this invariant as intermediate condition M to establish this condition at the start of our loop. The first abstract statement `statement1` is now refined with a refinement rule for assignments. We refine `statement1` to $\mathtt{i} = \mathtt{a.length} - 1;$ to start at the end of the array, and we verify that the Hoare triple $\{\mathsf{P}\}\mathtt{i} = \mathtt{a.length} - 1;\{\mathsf{M}\}$ is fulfilled with the concrete instances for P and M. In the example, the green border indicates a successful proof. The second abstract statement `statement2` is refined to a repetition statement using the invariant as discussed above. The loop guard is $\mathtt{a}[\mathtt{i}] \neq \mathtt{x}$. As long as the element `x` is not found, the loop is repeated. Here, we have to prove four conditions. First, the invariant must be established before the first loop iteration. Second, the postcondition P must follow after the last loop iteration. Third, the preservation of the invariant is shown in the last refinement, where we introduce the loop body $\mathtt{i} = \mathtt{i} - 1;$ to iterate through the array. Fourth, the loop must terminate. For termination, a variant is used which decreases monotonically and is bounded from below.

3.2 CorC

CorC [43] is a hybrid textual and graphical IDE to develop correct software using the CbC process. CorC supports programmers to refine programs and to check the correct application of the refinements. A check is for example on the correctness of a Hoare triple, the initial validity of a loop invariant, or the termination of a loop. For each check, CorC prepares a proof goal which is verified by the program verifier KeY [3] integrated in its backend. To be compatible with KeY, CorC prepares proof goals where the code is written in Java syntax with specifications in Java Dynamic Logic (JDL) [3]. The extent of language constructs covered in CorC is similar to the guarded command language [19] with statements for skip, assignment, function call, composition, selection, and

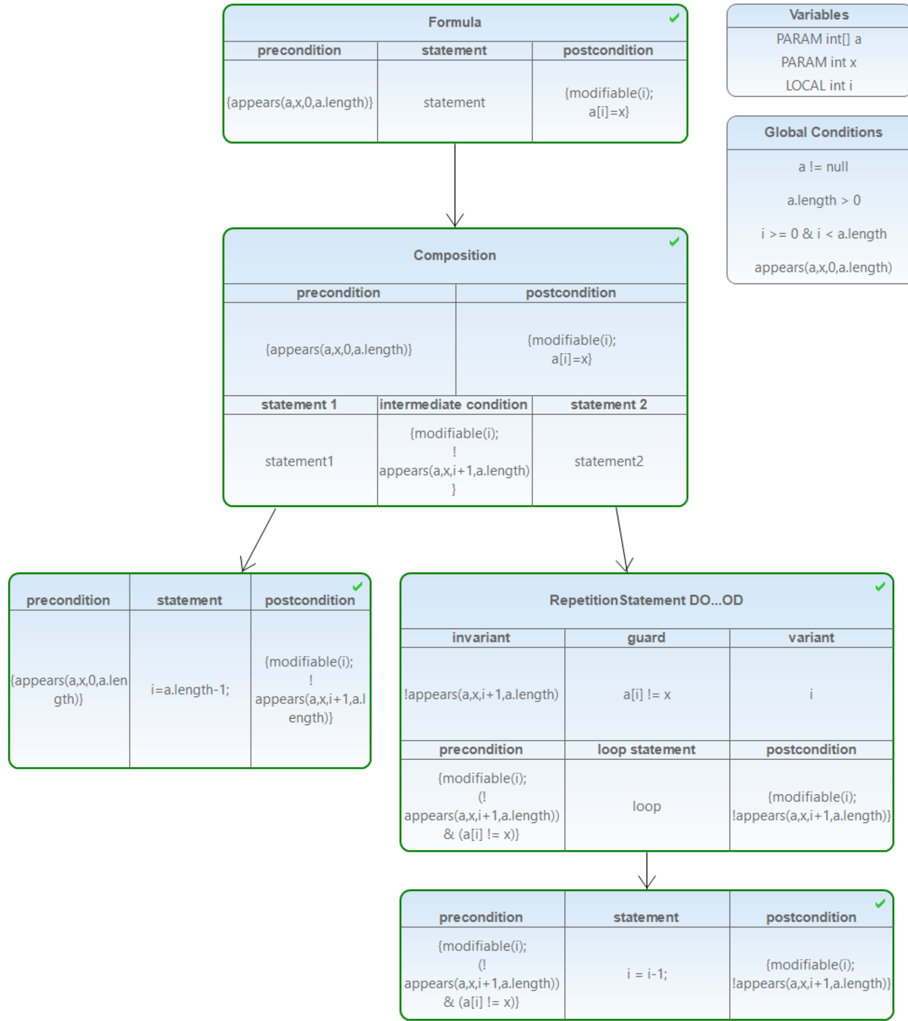

Fig. 1. Linear Search Algorithm Constructed in CorC

repetition. The CorC IDE also offers a textual editor. The syntax of the textual editor is based on Java that is enriched with keywords for the application of refinements and additional specifications for loop invariants and intermediate assertions. Programs created in the textual editor can be transformed to the graphical editor and vice versa.

In comparison to the verification of complete Java programs in KeY, CorC splits the verification effort of a complete method into the verification of several refinement steps (e.g., checking the refinement of introducing: skip, assignment, function call, composition, selection, repetition, weakening precondition, and

strengthening postcondition). In each step, a side condition, such as the establishment of a loop invariant before the first loop iteration, or the correctness of a Hoare triple, is verified. All proofs combined guarantee the correctness of the whole program. This split into several proofs can reduce the proof complexity and proof effort [54]. Thus, CorC can be used as a frontend for KeY that enables a correct-by-construction development process for the construction of individual algorithms.

4 Object-Oriented Development in CorC 2.0

Object-oriented programming is state-of-the-art in software engineering and supported by most modern programming languages. In the previous section, we described how single algorithms can be created using CorC. However, these algorithms are independent of any class structure which means that these single algorithms cannot access the same set of global fields like methods in a class in object-oriented programming can. Additionally, objects containing methods that have been created by CorC can only be created using laborious copy-and-paste workarounds. In other words, the current implementation of CorC can hardly be integrated into a software engineering process as it lacks a concept for modularization and ownership, as well as processes that enable the integration of CbC into a software development workflow. Therefore, in this section we present our concept for CbC-based object-oriented software development and the corresponding extension of CorC in the tool CorC 2.0. CorC 2.0 implements a roundtrip engineering from existing Java projects to CbC-based program development, which allows for a combination of post-hoc and CbC-based program development and verification.

4.1 Object-Oriented Concepts in CorC 2.0

Object-oriented programming is a common programming paradigm based around objects containing data fields and methods. In class-based languages, like Java, C++, C#, PHP, or Smalltalk, objects are instances of classes that have to be defined in advance. Classes are extensible templates for creating objects and provide initial values for fields and implementations of methods. Other paradigms that most object-oriented languages share are encapsulation, inheritance, polymorphism, and dynamic dispatch. As CorC already uses Java syntax in the refinement steps and KeY as deductive verification tool to verify the single refinement steps, we focus on object-orientation as realized in Java and how to combine this with CbC.

Classes. To support object-orientation in CorC 2.0, we introduce the construction of classes that hold methods implemented with CorC and fields that can be accessed by the methods contained in the respective class. The visibility of fields can be modified using the Java visibility modifiers `public`, `private`, `package`, and `protected`. Fields can also be defined as `static` or

`final`. Besides fields that have been defined in the class, methods of that class can define a set of local variables including parameters and a return variable. Additionally, we add class invariants as a new specification type to our class definitions. Class invariants specify conditions that are fulfilled by the class, i.e. that are preserved by all methods of that class or re-established at the end of method execution. To guarantee that a method created with CorC fulfills the class invariants, they are automatically added to the pre- and postcondition of the starting Hoare triple when that method is constructed.

Method Calls. Methods can either be called inside of the same class, by an object instantiating the class, or directly by the class if the method is static. The implementation of that method can either be in CorC or in Java. For the verification of a method call, CorC supports inlining and contracting [28] (i.e., inserting its implementation or using its contract as defined in the CbC method call refinement rule). When contracting is used, it is assumed that the contract holds for that method, however, this is not specifically verified in this step.

Framing. Besides a pre- and a postcondition, a frame that contains all variables whose data can be modified is defined for each method. This information helps callers of the method to determine which parts of the state are not changed due to the call [12]. For formal verification of object-oriented programs, framing is important, because the caller implicitly knows which fields remain unchanged during the execution of a method [3, 55]. Furthermore, framing is important for information hiding [30] and to avoid unwanted side effects [32]. In CorC, the frame of a method is automatically determined by traversing its refinement steps and collecting the variables that are on the left of an assignment. However, the frame can also be defined manually by the user. For the verification of a method with frame, it is checked whether all variables that are not included in the frame still have the same value as before the execution of that method.

Inheritance and Interfaces. Inheritance and interfaces are two important features in Java. While interfaces can be used as a layer of abstraction, inheritance can be used to create classes built upon existing classes to, for example, enable code reuse. For both, inheritance and interfaces, we check that the Liskov principle is fulfilled by the child class or the class that implements the interface. This means that class invariants that are defined in the parent class or the interface also have to be fulfilled by the class that extends or implements it. Furthermore, if a method is overridden in the child class or implemented from an interface it also has to fulfill the contract that has been defined for that method in the parent class or interface. We do not require an interface for every class.

Limitations. Besides methods, classes also contain constructors that are used to instantiate an object from a class. Even though constructors are very important in object-oriented programming, we do not support their creation and

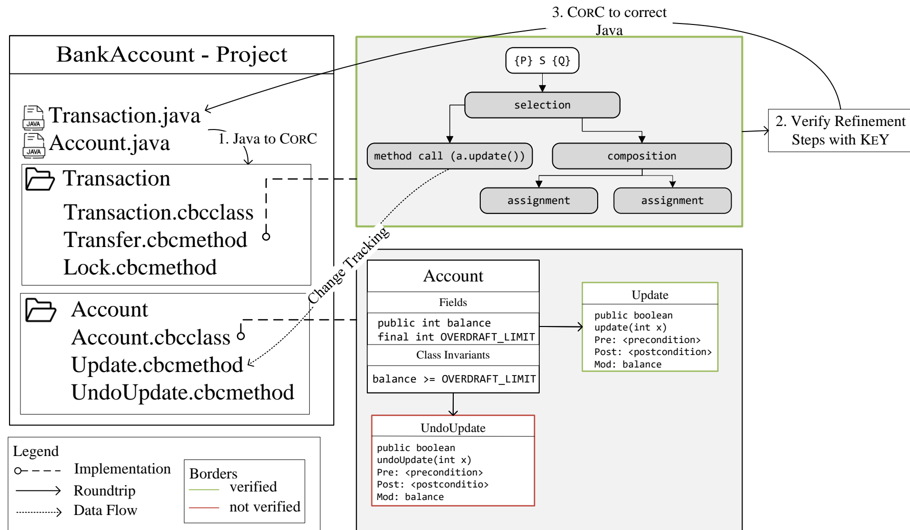

Fig. 2. Development Process in CorC 2.0

verification in CorC. However, an initialization method which creates a new instance by calling a default constructor can be created and verified with CorC. The contracts of methods are limited to a pre- and postcondition pair including a frame. Exceptional behavior such as expecting a specific exception to be thrown cannot be expressed in the contracts, and hence we cannot reason about exceptional behavior.

4.2 Development Process in CorC 2.0.

In Fig. 2, we give an overview of the project structure and the development process in CorC 2.0. We do this using an example of a Bank Account software system that consists of two classes. The class `Account` has two methods `update` and `undoUpdate` to manipulate the balance of the account. The class `Transaction` provides the method `transfer` which allows to transfer money from a source account to a destination account and the method `lock` which locks an account such that the balance cannot be changed anymore.

On the left side of Fig. 2, we show the project structure of the Bank Account system. There are two folders named by the two classes `Account` and `Transaction` that hold all files that have been created with CorC for each class. The cbcclass and cbcmethod files are representations of the Java classes and methods in CbC format which can be displayed, edited, and verified by CorC. Each folder contains a cbcclass file which is also named after the class and contains all information related to this class (i.e., class and method information). The implementation of `Account.cbcclass` is shown in the bottom center of Fig. 2 similar to a UML class diagram. There is one bigger box with the title

Account that defines the field `balance` and the constant `OVERDRAFT_LIMIT` and a class invariant. If this class inherits from another class or implements an interface this would be defined using the Java keywords `extends` and `implements`. The methods `update` and `undoUpdate` are shown in two separate boxes that are connected to the Account class. They show the method signature and the contract consisting of precondition, postcondition, and framing. Furthermore, their border is either green or red to display their verification status.

Besides the cbcclass files, there is also a cbcmethod file per method in the class folders. The development of methods generally stays the same as before in CorC without object-orientation. The content of `Transfer.cbcmethod` is shown in the top center of Fig. 2. It shows the refinement steps that are used in CorC to construct method `transfer` starting from the starting Hoare triple $\{\mathsf{P}\}\ \mathsf{S}\ \{\mathsf{Q}\}$. Precondition P and postcondition Q are the same as the pre- and postcondition that are contained in `Transaction.cbcclass` for method `transfer`. The single refinement steps are created in CorC and verified with KeY as described in Section 3.

In the following, we describe some of the new core features of CorC 2.0, which are important for the integration of CbC into the software engineering development process.

Roundtrip Engineering. To simplify the integration of CorC 2.0 into existing Java software system development, we introduce a roundtrip engineering functionality. This roundtrip engineering process can be used for example for (1) implementing new methods using CbC, (2) guaranteeing the correctness of an already implemented method, or (3) to be able to better track the source of error when the developer fails to prove a certain method with post-hoc verification. Either way, the correct and changed implementation and specification can be integrated back into the original project. The roundtrip engineering is performed in three steps as displayed in Fig. 3. These three steps can be iteratively repeated to create an incremental development process.

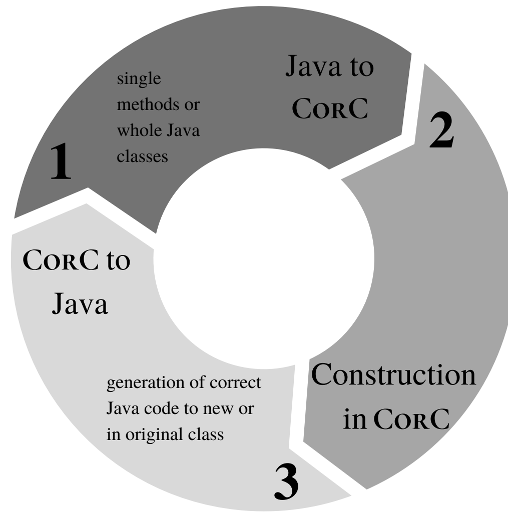

Fig. 3. Roundtrip Engineering Workflow

Step 1: If there are already Java classes that contain methods that need to be verified, the classes and methods can be converted into the cbcclasses and cbcmethods in a first step. During this step, the user can select the methods

that shall be converted. The user can then complete missing specifications and annotations (e.g. intermediate conditions, loop invariants, variants, ...) in the cbcmethods. The cbcclasses and cbcmethods can also be created from scratch in CorC, which makes this first step optional. In that case, the user has to manually apply the refinement rules to construct the methods in Step 2.

Step 2: The refinement steps in the cbcmethods are verified. Method calls can either be verified by inlining or contracting (i.e. using either implementation or contract of the called method). If a method call is verified using its contract, the method can either be implemented in CorC or provided in Java specified with a JML contract. This allows the user to freely combine CorC with existing Java methods and post-hoc verification. However, in that step it is not verified whether a called method actually fulfills its contract. Consequently, not all methods need to be specified (when inlining is used) and methods that are specified do not necessarily have to be verified to be called and can be assumed to be correct.

Step 3: CorC can generate correct Java code from the verified cbcmethods, either to a new Java class or back into the original Java class it has been imported from where it replaces the original implementation and contract.

Change Tracking. To further improve the usability of CorC, we introduce a notification system that keeps track of changes by setting already verified refinements to not verified. This is especially critical for methods using method calls, inheritance, or interfaces. In Fig. 2, the implementation of method `transfer` calls method `update` on `Account a`. Now, if there is a change in method `update` in class `Account`, the method call refinement in method `transfer` needs to be re-verified, as the old proof relies on a possibly outdated contract. The change tracking system prevents the developer from overlooking these changes. Additionally, it enables the direct navigation to the affected method (in our example to method `transfer`) for re-verification. In the background, the affected refinement rules are automatically set to not verified so that these refinement steps cannot mistakenly be assumed to be correct. Since CbC has a fine-grained structure with single refinement steps, not all refinement steps of a method have to be re-verified, but only those that are affected by the change. CorC 2.0 can better maintain the correctness of evolving software than its predecessor, since it is no longer possible to have refinements falsely marked as verified.

4.3 Implementation

CorC 2.0[6] is an open-source Eclipse plug-in supporting the development of object-oriented programs using CbC as described in this work. CorC captures the structure of a CbC program including the refinement rules through two meta-models, one for the class files and one for the methods, modeled using the Eclipse Modeling Framework[7]. The graphical editing framework Graphiti[8] is

[6] https://github.com/TUBS-ISF/CorC
[7] https://eclipse.org/emf/
[8] https://eclipse.org/graphiti/

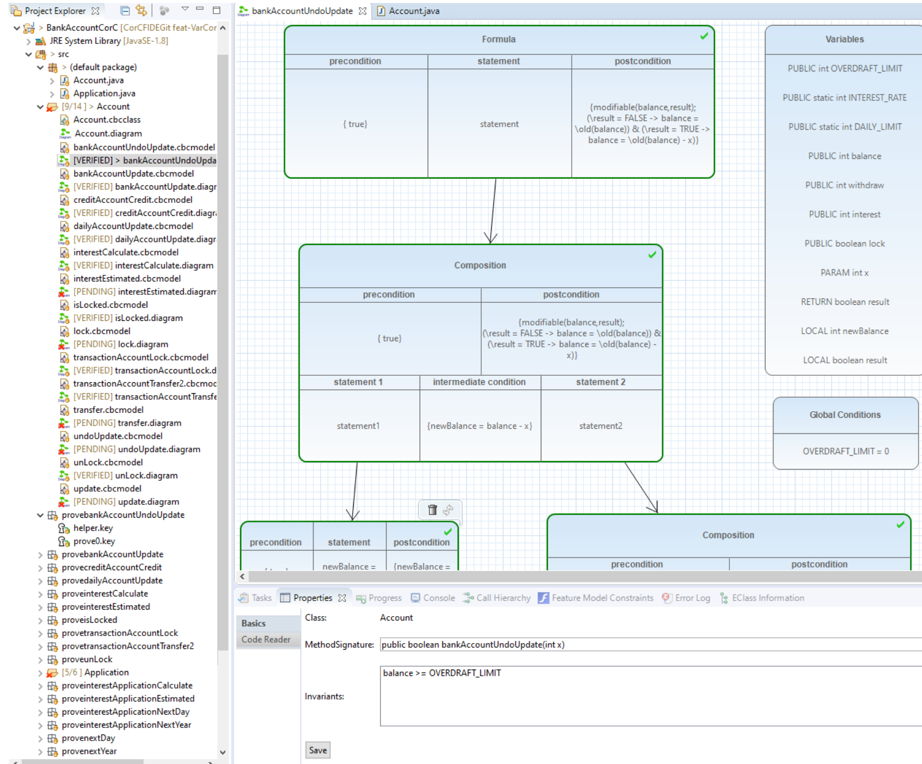

Fig. 4. Screenshot of CorC 2.0 with Method `update`

used to visualize the underlying meta-models. For the methods, we use a tree-like structure. The beginning of a method in CbC always consists of a Hoare triple, which can be refined until there are no more abstract statements. Thereby, the pre- and postconditions and other annotations are automatically passed on in each refinement step. Furthermore, the deductive verification tool KeY [3] is used to prove the correct usage of each refinement rule.

In Fig. 4, we show a screenshot of the graphical view to develop methods in CorC 2.0. On the left side, there is the project structure of the Bank Account case study which has been used as an example in the previous subsection. The Java-classes are in the default package. Then, there are folders named after the classes holding the information about the class and all methods in form of diagram and model files named after the distinct classes and methods. The diagram files (*<methodName/className>.diagram*) contain the graphical representation and the model files (*<methodName>.cbcmodel/<className>.cbcclass*) store the information about the methods and classes in the corresponding meta-model. The *prove<methodName>* folder stores generated proof files, which contain the side conditions for a refinement step which need to be verified. Each proof file is verified (semi-)automatically by KeY. For a better overview, in front of the

folder name the proportion of verified methods is given and in front of the <*methodName*>*.diagram* files the verification state is given. In this context, *verified* means that all refinement steps of a method could be proven and *pending* means that at least one refinement step is still unverified.

In the center of Fig. 4, we can see the CorC-diagram of method `undoUpdate`. At the top of the diagram, we can see a formula component for our starting Hoare triple. Underneath, we see a refinement step using the composition refinement rule. That refinement rule splits the abstract statement into two abstract statements. Afterwards, these are further refined. We can right-click on these components to trigger a verification process. In the verification process, we translate the Hoare triple of the selected component into a proof file for the corresponding refinement rule in the format required by KeY. All components in the CbC construction tree are marked green in our example so that we can conclude that all refinement steps have been verified. If a refinement step could not be verified, the corresponding component is marked red which enhances the traceability of incorrectly defined refinement rules or inconsistencies in relation to the specifications. At the top right, we can see two boxes which hold the defined variables and global conditions.

At the bottom of Fig. 4, we can see the properties view with the currently active *Basics* tab. It shows further information about the method `undoUpdate`, such as the class it belongs to, its signature, and the class invariants it has to fulfill. The signature of the method can also be edited. To change any other information, the class file has to be opened. The other tab in the properties view is called *Code Reader* and displays the selected Java statements or specifications in the diagram in a bigger window with syntax highlighting. It enables better readability of especially long specifications and helps to modify them more easily without introducing syntax errors.

In Fig. 5, we show a screenshot of the class view in CorC 2.0. It displays the content of file *Account.diagram*. In the top center, we see a component for the class definition that looks similar to a UML class diagram. At the top, it displays the name of the class, and below that, it lists the class invariants and fields with their visibility, type, and name. Around the class component, there are several other components which are the methods that are implemented in this class. They show the signature and the contract of each method. Additionally, their verification status is displayed by the red and green borders. In this view, new methods, fields, and class invariants can be added and existing ones can be edited. Changed information, for example a changed precondition of a method or a changed type of a field, is directly available in all method diagrams since they directly reference this file. In the case of a changed precondition, the user is notified by the change tracking mechanism and the verification status is set to not verified as described in the previous subsection. For an easy navigation to the method diagrams, the user can double-click on a method component.

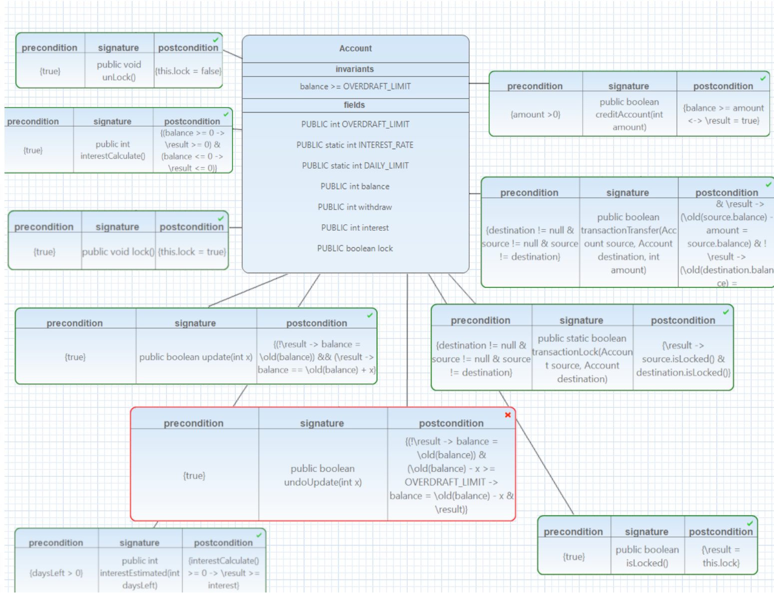

Fig. 5. Screenshot of Class `Account` in CorC 2.0

5 Evaluation

In this section, we evaluate CorC 2.0 by comparing it with post-hoc verification. We use KeY for post-hoc verification, since KeY is also the integrated verifier in CorC, but KeY can be understood as synonym for post-hoc verification. To evaluate whether it is feasible to construct correct programs with CorC, we want to answer the following two research questions:

RQ1: How does the verification effort (w.r.t. execution time and proof steps) of verifying algorithms in CorC compare to post-hoc verification in KeY?

RQ2: How does the CorC development process assist in creating correct programs in comparison to the assistance of KeY for post-hoc verification?

The first research question is answered by creating three case studies with both CorC and KeY, and measuring the verification time and the number of verification steps. Each case study consists of several Java classes containing specified methods. Each method is created and verified with CorC. For the post-hoc verification approach, we verified the methods written in Java and specified with JML (precondition, postcondition, and loop invariants, but no further intermediate specification was given). For method calls, we used contracting to prove correctness, but inlining can be used as an alternative. We always used KeY as automatic verification tool. We measured the verification steps by the number of rule applications of KeY. The verification time was measured five times and the average was calculated. We do not consider the manual effort of

Case Study	**#Classes**	**#Methods**	**#Verified with CorC**	**#Verified with KeY**
Bank Account [52]	2	10	10	9
Email [24]	2	12	12	8
Elevator [39]	5	18	18	17

Table 1. Metrics of the case studies

writing additional specification for CorC. For the second research question, we qualitatively discuss the CorC tool by referring to two user studies conducted at TU Braunschweig. We also discuss the new features of CorC 2.0.

Case Studies. We have three cases studies that are implemented and verified with CorC and KeY. We decide to use the case studies *Bank Account* [52], *Email* [24], and *Elevator* [39] because they are implemented in an object-oriented fashion, such that we can evaluate the new feature of CorC 2.0. In Table 1, we show some metrics for the case studies. We have two to fives classes and 10 to 18 methods per case study. The size of the methods ranges from 1 to 20 lines of code.

5.1 RQ1 - Verification Time and Verification Steps

To answer the first research question, we verified all 40 methods in CorC. For post-hoc verification, we used the same pre-/postcondition specifications as in CorC. However, six algorithms could not be verified (cf. Table 2). The verification of methods with KeY failed, for example, in the steps where the loop invariant must be proven. In CorC, the verification of loops is split into several smaller proofs reducing the proof complexity. Another reason for a reduced proof complexity in CorC is that we introduce intermediate specifications, which guide the verifier. In contrast, applying post-hoc verification is more coarse-grained, as only precondition, postcondition, and loop invariants are specified. We observed that debugging verification problems that could not be verified automatically in post-hoc fashion, is more challenging than debugging the same problem constructed with CbC in CorC.

In Figure 6, we show the average verification time measured in milliseconds, and in Figure 7, we show the average verification steps for each case study, but only for the 34 methods which could be verified with CorC and KeY. The verification time ranges from 0.1 seconds for some methods to 16 seconds for the most complex methods. For 22 methods, the verification time with CorC is faster, for 12 methods the verification with KeY is faster. The largest differences are: `enterElevator` is 268% faster with CorC, `addWaitingPerson` is 271% faster with KeY. The average verification time for the *Email* case study shows that the verification is 23% faster with CorC, but on average the *Elevator* case study is 24% faster with KeY. For the number of proof steps, we got similar

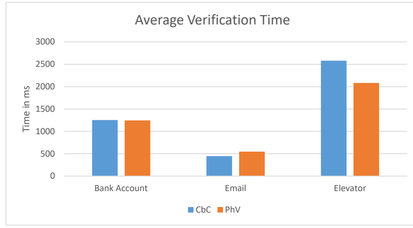

Fig. 6. Average Verification Time of the Case Studies with CbC and PhV

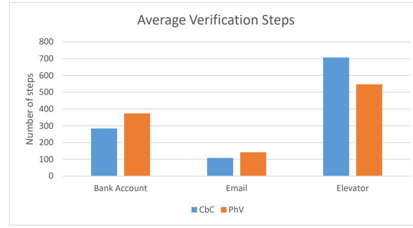

Fig. 7. Average Verification Steps of the Case Studies with CbC and PhV

results. In 26 cases, CorC has fewer steps, and in 8 cases, KeY has fewer steps. The case studies *Bank Account* and *Email* are on average 32% and 31% smaller with CorC, and *Elevator* is 29% smaller with KeY. Overall, the results are of the same order of magnitude. The exact verification time and steps for each method are shown in Appendix A.

Discussion. Regarding the verification effort, no trend can be identified. Therefore, we cannot answer the research question positively that the verification with CbC is faster than with PhV in terms of verification time or verification steps. The verification time and the verification steps are similar for both approaches. However, we could verify more methods in total with CorC. The additional specifications in the form of intermediate annotations and the splitting into several smaller proofs facilitates the completion of proofs, but it does not significantly affect the verification effort. As we are promoting to use CbC and PhV in concert, a similar verification effort is beneficial. There is no discernible disadvantage in terms of effort in using one of the two approaches.

5.2 RQ2 - Usability of CorC

For the second research question, we conducted two user studies [42, 45] with a total of 23 students from the TU Braunschweig. In both studies, a group of students was divided into half. One group created and verified a method with CorC and a second method with KeY. The second group created the same methods, but used the tools in reverse order. We measured the defects in the developed and verified methods and performed a questionnaire on the usability of CorC and KeY at the end of the user study. The first user study took place in person in 2019 [45], the second user study was online in 2021 [42]. In both user studies and with both tools, we had several correct implementations but which were not verified in the given time frame. A common problem was a too weak loop invariant. In the usability questionnaire, most participants preferred CorC over KeY due to the better and more fine-grained feedback when errors occurred during the verification. It was easier to detect and solve errors with CorC. A minority of participants preferred KeY because they were more familiar with the syntax of Java and JML. As we are now supporting roundtrip development

with CorC 2.0, we believe that this statement is weakened. Users can now freely develop and verify programs with CorC or KeY and generate the program for the other approach automatically. Thus, the preferred tool can be used without restrictions.

During the construction of the three case studies (cf. RQ1), CorC's change tracking feature was valuable. When we verified a refinement step that called another method, it occurred that we had to change the contract of this called method. The change tracking feature then set the affected method calls in all CorC programs to not be verified. With this feature, we did not miss any open verification obligations. In summary, we can confirm that CorC assists in developing correct software. Additionally, with the new features of CorC 2.0, the implementation of object-oriented code is supported.

5.3 Threats to Validity

External Validity. The methods implemented in the three case studies have a size of 1 to 30 lines of code. These small methods reduce the generalizability for larger algorithmic problems. While CorC 2.0 extends the application field to object-orientation, we still consider CorC to be a tool for smaller, but challenging algorithmic problems. The generalizability of the user studies are limited as only 23 students participated, but due to the small number of participants, we were able to interview them in more detail. Nevertheless, the participants were no experts in verification. We classify the participants as junior developers.

Internal Validity. We wrote most of the specifications of the three case studies ourselves. Thus, there could still be defects in the specifications or implementations. However, we were able to verify all methods with CorC and most methods with KeY, which is a strong indication of correctness of the case studies. In particular, we checked the equality of the specifications for the two different approaches. The time frame of the user study was limited: the participants had only 30 minutes for each method implementation. With more time, we expect that more participants verify the assigned methods.

6 Beyond Monolithic Program Construction with CorC

The previous sections emphasize our ongoing vision to integrate the correctness-by-construction methodology into the realm of mainstream software development through sophisticated tool support. Besides extending CorC's programming model with advanced paradigms, such as object orientation (cf. Sec. 4), a natural follow-up is to develop concepts and tool support that make the correct-by-construction approach available *at scale*. Currently, we aim to address this by two further directions. First, VarCorC is a framework for CbC-based development of *variational* software. That is, instead of developing monolithic programs, the goal is to systematically construct a family of similar software programs following the CbC paradigm. Second, we study the role of correct-by-construction

implementations in software architectures with ArchiCorC, where the main goal is to *bundle* CorC programs into *reusable software components*. We briefly present our vision for both lines of research in the following.

VarCorC: Correct Variational Software Construction

Software product lines [40] are increasingly used to lower the costs in producing custom-tailored software products, also including the domain of safety-critical software systems. VarCorC is an extension of CorC to create feature-oriented software product lines [11]. Software product lines are families of related programs sharing a common code base [18]. The common and varying parts of a product line are defined by features. In feature-oriented programming [5, 10], features are implemented by feature modules. Similar to inheritance in object-oriented languages, in a feature module, methods can be added or overridden in a specific order defined by the product line. Overridden methods can use the original keyword to call the previous implementation of that method. The feature configuration then defines the explicit relationship between feature modules. For VarCorC, we extended CbC by two new *variability-aware refinement rules*, one for original calls and one for variational method calls. Since implementation of a method depends on the different features, method calls encompass variability in software product lines. Additionally, we integrated the concept of *contract composition* [51] to allow for a varying method contract per feature that can be composed along a feature configuration to form the contract of a method in a product of the product line.

ArchiCorC: Correct Software Architectures

The benefits of combining correctness-by-construction with *component-based software engineering* [16, 46, 49] are manifold. For instance, component-based architectures allow to establish a repository of correct-by-construction components. This is not only interesting for standard libraries, where implementations are accessed through explicit interfaces, but also for third-party developments that are easier to integrate into personal projects. Most importantly, modularization of correct implementations into components allows developers to think about how to *compose* software systems instead of how to program a monolithic software system from scratch. We argue that this is the foundation for building large and complicated systems that are based on the correctness-by-construction approach.

As an extension to CorC, we propose a framework and an open-source implementation named ArchiCorC [27] that connects UML-style component modeling, specification, and code generation. In more detail, ArchiCorC comprises four key ingredients. First, a *component and interface description language* is used to describe interconnections between provided and required interfaces of components, where interfaces comprise method signatures that are specified with Hoare triples. Second, a *construction technique* aids developers with either refining method signatures of provided interfaces to correct implementations using

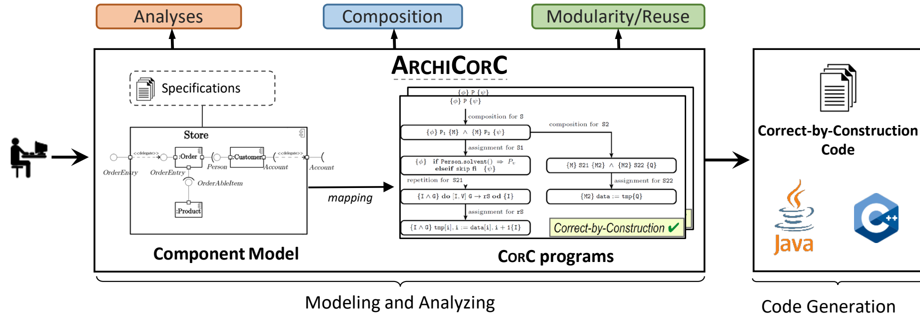

Fig. 8. Envisioned Workflow of the ArchiCorC Development Process

CorC itself, or with mapping signatures to already existing CorC programs. Third, ArchiCorC integrates analyses and algorithms to check compatibility between components and to build composites to form larger components. Finally, ArchiCorC allows to generate code in a general-purpose programming language (e.g., Java).

In Figure 8, we illustrate an envisioned development process using ArchiCorC. A developer starts by designing a high-level component model including required and provided interfaces and connections between them. Hierarchical composition allows to build *larger* components from a set of smaller ones. Method signatures of provided interfaces of atomic components (i.e., components that are not decomposed any further) must be mapped to CorC programs, which are assumed to be correct-by-construction. The component model can then be translated to a general-purpose programming language (e.g., Java or C++) and can be imported into other projects. This development process embodies the next milestone of our ongoing vision of correct-by-construction software development.

7 Conclusion

Our long-term vision is to make the correctness-by-construction (CbC) approach accessible for large-scale software development. The formal framework of CbC enables developers to start with a specification for safety-critical parts and algorithms. Then, the framework guides developers in deriving a provably correct implementation. A major stepping stone towards this vision is the development of tool support that allows to apply traditional software development and CbC in concert. In this work, we presented state-of-the-art tool support for this mission, namely the CorC 2.0 tool family, including a comprehensive overview of its current status and prospective directions.

In particular, we introduce CorC 2.0 (the successor of CorC), which combines CbC with object-oriented software development in Java. In CorC, single algorithms are developed completely independently of other programs, making

the integration into software engineering processes impractical. As a new key feature, CorC 2.0 adds a class concept for structuring programs (i.e., methods) inspired by Java's object-oriented programming model. CorC 2.0 now supports a roundtrip engineering process that is important for applying CbC to existing software projects; existing Java classes comprising specified methods can be converted into CorC 2.0 projects and, after verified construction, they can be converted back to verified Java implementations. This strong improvement in tool support directly targets scalability concerns of the CbC approach, as it allows developers now to decide whether parts are (1) verified using post-hoc verification techniques, (2) implemented following the CbC approach, or (3) remain unverified.

In alignment with our vision, we believe that CorC 2.0 allows developers to address program verification of general-purpose programming languages in a new way. One outcome of our evaluation illustrates that CorC 2.0 can sometimes outperform post-hoc verification with respect to verification effort and success rate. Although specification effort can be higher, the benefits of additional specification together with the fine-grained refinement rules are easier debugging and better feedback in general. Our future direction with the CorC ecosystem is therefore a seamless integration into mainstream software development. One focal point is to identify and study possible synergies when applying post-hoc verification and CbC in concert. Another focal point is to conduct larger user studies, which provide important insights on how CbC is applied in practice and, consequently, influence the development of the CorC ecosystem in general. Only the recent advancements made in CorC 2.0 enable us to develop more complex case studies necessary to address these future directions.

Acknowledgements

We thank Maximilian Kodetzki from TU Braunschweig for implementing large parts of the new features for CorC 2.0.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. 1st edn. (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer 12(6), 447–466 (2010)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification – The KeY Book (2016)
4. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of Concurrent Systems with VerCors. In: SFM. LNCS, vol. 8483, pp. 172–216. Springer (2014)
5. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines (2013)

6. Back, R.J.: Invariant Based Programming: Basic Approach and Teaching Experiences. Formal Aspects of Computing 21(3), 227–244 (2009)
7. Back, R.J., Eriksson, J., Myreen, M.: Testing and Verifying Invariant Based Programs in the SOCOS Environment. In: International Conference on Tests and Proofs. pp. 61–78. Springer (2007)
8. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer Science & Business Media (2012)
9. Barnes, J.G.P.: High Integrity Software: The Spark Approach to Safety and Security. Pearson Education (2003)
10. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement 30(6), 355–371 (2004)
11. Bordis, T., Runge, T., Schaefer, I.: Correctness-by-Construction for Feature-Oriented Software Product Lines. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 22–34 (2020)
12. Borgida, A., Mylopoulos, J., Reiter, R.: On the Frame Problem in Procedure Specifications. IEEE Transactions on Software Engineering 21(10), 785–798 (1995)
13. Bruns, D., Klebanov, V., Schaefer, I.: Verification of Software Product Lines with Delta-Oriented Slicing. pp. 61–75 (2011)
14. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
15. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. pp. 472–479 (2011)
16. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering 37(5), 593–615 (2010)
17. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: SEFM. LNCS, vol. 7504, pp. 233–247. Springer (2012)
18. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications (2000)
19. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM 18(8), 453–457 (1975)
20. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall PTR, 1st edn. (1976)
21. Gries, D.: The Science of Programming. 1st edn. (1981)
22. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Component-based Synthesis Applied to Bitvector Programs
23. Hähnle, R., Schaefer, I.: A Liskov Principle for Delta-Oriented Programming. pp. 32–46 (2012)
24. Hall, R.J.: Fundamental Nonmodularity in Electronic Mail. Automated Software Engineering 12(1), 41–79 (2005)
25. Heisel, M.: Formalizing and implementing Gries' program development method in dynamic logic. Science of Computer Programming 18(1), 107–137 (jan 1992)
26. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Asian Symposium on Programming Languages And Systems. Lecture Notes in Computer Science, vol. 6461, pp. 304–311. Springer (2010)
27. Knüppel, A., Runge, T., Schaefer, I.: Scaling Correctness-by-Construction. In: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece. vol. 12476, pp. 187–207. Springer (2020)
28. Knüppel, A., Thüm, T., Pardylla, C.I., Schaefer, I.: Scalability of Deductive Verification Depends on Method Call Treatment. pp. 159–175 (2018)

29. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming (2012)
30. Leavens, G.T., Müller, P.: Information Hiding and Visibility in Interface Specifications. pp. 385–395 (2007)
31. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
32. Leino, K.R.M., Nelson, G.: Data Abstraction and Information Hiding. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(5), 491–553 (2002)
33. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Transactions on Programming Languages and Systems (TOPLAS) 2(1), 90–121 (jan 1980)
34. Meyer, B.: Eiffel: A Language and Environment for Software Engineering. Journal of Systems and Software 8(3), 199–246 (1988)
35. Meyer, B.: Applying Design by Contract 25(10), 40–51 (1992)
36. Morgan, C.: Programming from Specifications. Prentice Hall (1998)
37. Oliveira, M., Cavalcanti, A., Woodcock, J.: ArcAngel: A Tactic Language for Refinement. Formal Aspects of Computing 15(1), 28–47 (2003)
38. Pearce, D.J., Groves, L.: Whiley: A Platform for Research in Software Verification. In: SLE. LNCS, vol. 8225, pp. 238–248. Springer (2013)
39. Plath, M., Ryan, M.: Feature Integration Using a Feature Construct. Science of Computer Programming 41(1), 53–84 (2001)
40. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques (2005)
41. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program Synthesis from Polymorphic Refinement Types. ACM SIGPLAN Notices 51(6), 522–538 (aug 2016)
42. Runge, T., Bordis, T., Thüm, T., Schaefer, I.: Teaching Correctness-by-Construction and Post-hoc Verification–The Online Experience. In: Formal Methods Teaching Workshop. pp. 101–116. Springer (2021)
43. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool Support for Correctness-by-Construction. In: FASE. pp. 25–42. Springer (2019)
44. Runge, T., Servetto, M., Potanin, A., Schaefer, I.: Traits for Correct-by-Construction Programming. To be published (2021)
45. Runge, T., Thüm, T., Cleophas, L., Schaefer, I., Watson, B.W.: Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In: Refine. Springer (2019)
46. Sametinger, J.: Software Engineering with Reusable Components. Springer Science & Business Media (1997)
47. Steinhöfel, D., Hähnle, R.: Abstract Execution. In: International Symposium on Formal Methods. pp. 319–336. Springer (2019)
48. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) pp. 341–355 (1994)
49. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Pearson Education (2002)
50. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines 47(1), 6:1–6:45 (2014)
51. Thüm, T., Knüppel, A., Krüger, S., Bolle, S., Schaefer, I.: Feature-Oriented Contract Composition 152, 83–107 (2019)

52. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-Based Deductive Verification of Software Product Lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. p. 11–20. GPCE '12, Association for Computing Machinery, New York, NY, USA (2012)
53. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9035, pp. 566–580. Springer (2015)
54. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-Construction and Post-Hoc Verification: A Marriage of Convenience? In: International Symposium on Leveraging Applications of Formal Methods. pp. 730–748. Springer (2016)
55. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic, and Predicate Abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)

A Appendix

Method	#Steps CbC	#Steps PhV	Time in ms CbC	Time in ms PhV
undoUpdate	205	183	730,75	747,3
update	181	188	626,75	627,67
creditAccount	25	45	104,5	132,67
dailyUpdate	467	636	1990,5	2261
interestCalculate	422	Unclosed	1164	Unclosed
transactionLock	432	663	2253	1995,67
transactionTransfer	799	989	3556,75	3323,3
unLock	21	41	104,5	175,67
interestNextDay	233	381	1066,25	1154,67
interestNextYear	191	239	836,75	786,3
constructClient	50	68	207,75	292,3
createClient	1391	Unclosed	7108,5	Unclosed
getClientByAdress	1420	Unclosed	4509,75	Unclosed
getClientById	67	Unclosed	210,75	Unclosed
outgoing	67	61	212,75	200
resetClients	673	Unclosed	2450,75	Unclosed
constructEmail	36	42	111,25	156,33
createEmail	558	772	2591	2996,67
setEmailBody	38	47	110,25	185
setEmailFrom	40	51	104,5	192,33
setEmailSubject	38	47	113,25	170,67
setEmailTo	38	47	111,5	173,67
areDoorsOpen	64	41	229,25	141,67
buttonIsPressed	70	69	206	144
enterElevatorBase	805	1092	4829,5	4577
enterElevator	457	2315	2370	8710
pressButtonBase	87	94	317,25	529
pressButton	97	109	444,25	503,33
resetFloorButton	79	84	336,25	287
reverse	192	108	985,25	338,33
stopRequestedBase	878	1150	3179	4367,67
createEnvironment	1367	Unclosed	4375,5	Unclosed
isTopFloor	86	97	255,5	345,33
addWaitingPerson	4768	1041	16012,25	4320,33
callElevator	28	42	105,5	174,67
createFloor	346	1066	1641,75	4027
reset	29	42	106,25	173,67
createPerson	3835	1764	12323	6171,67
PersonenterElevator	163	134	432	393,67
PersonleaveElevator	30	44	105,25	174,67

Table 2. Verification Time and Verification Steps of All Methods