Logical Methods in Computer Science
Volume 19, Issue 2, 2023, pp. 16:1–16:36
https://lmcs.episciences.org/

Submitted Nov. 29, 2022
Published Jun. 07, 2023

FLEXIBLE CORRECT-BY-CONSTRUCTION PROGRAMMING

TOBIAS RUNGE [a,b], TABEA BORDIS [a,b], ALEX POTANIN [d], THOMAS THÜM [e], AND INA SCHAEFER [a,b,c]

[a] Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology, Germany
e-mail address: {tobias.runge, tabea.bordis, ina.schaefer}@kit.edu

[b] Institute of Software Engineering and Automotive Informatics, TU Braunschweig, Germany

[c] School for Data-Science and Computational Thinking, Stellenbosch University, South Africa

[d] School of Computing, Australian National University, Australia
e-mail address: alex.potanin@anu.edu.au

[e] Institute of Software Engineering and Programming Languages, University of Ulm, Germany
e-mail address: thomas.thuem@uni-ulm.de

Abstract. Correctness-by-Construction (CbC) is an incremental program construction process to construct functionally correct programs. The programs are constructed stepwise along with a specification that is inherently guaranteed to be satisfied. CbC is complex to use without specialized tool support, since it needs a set of predefined refinement rules of fixed granularity which are additional rules on top of the programming language. Each refinement rule introduces a specific programming statement and developers cannot depart from these rules to construct programs. CbC allows to develop software in a structured and incremental way to ensure correctness, but the limited flexibility is a disadvantage of CbC. In this work, we compare classic CbC with CbC-Block and TraitCbC. Both approaches CbC-Block and TraitCbC, are related to CbC, but they have new language constructs that enable a more flexible software construction approach. We provide for both approaches a programming guideline, which similar to CbC, leads to well-structured programs. CbC-Block extends CbC by adding a refinement rule to insert any block of statements. Therefore, we introduce CbC-Block as an extension of CbC. TraitCbC implements correctness-by-construction on the basis of traits with specified methods. We formally introduce TraitCbC and prove soundness of the construction strategy. All three development approaches are qualitatively compared regarding their programming constructs, tool support, and usability to assess which is best suited for certain tasks and developers.

Key words and phrases: Traits, Correctness-by-Construction, Formal Methods, Post-hoc Verification.

1. Introduction

Correctness-by-Construction (CbC) [Dij76, Gri87, KW12, Mor94] is a methodology in the field of formal methods to incrementally construct functionally correct programs guided by a pre-/postcondition specification.[1] In contrast to post-hoc verification, where a program is typically specified and verified after implementing it, CbC is based around successively creating a program together with the specification. This is achieved by applying refinement rules from a small set of defined rules where in each refinement step, an abstract statement (i.e., a hole in the program) is refined to a more concrete implementation that can still contain some nested abstract statements. While refining the program, the correctness of the whole program is guaranteed through applicability conditions that are defined in the refinement rules. The construction ends when no abstract statement is left.

The underlying idea of this specification-first, refinement-based approach is that developers are forced to think about their algorithm more thoroughly rather than having a trial-and-error verification approach. This trial-and-error verification can oftentimes be experienced with post-hoc verification because programs are implemented first and therefore not well-structured for the verification process which leads to tedious verification work. Additionally, through the structured reasoning discipline that is enforced by the refinement rules in CbC, errors are more likely to be detected earlier in the design process, and it is argued that program quality increases and verification effort is reduced [KW12, WKSC16].

Despite these benefits, CbC intuitively has a drawback: The flexibility of creating a program is limited to the set of refinement rules and the rigid, rule-based construction process of applying one rule at a time. This is even increased by the granularity of the rules which explicitly only allow to use one language construct at a time (e.g., one assignment to a variable). Additionally, the refinement rules extend the programming language (i.e., refinement rules are an additional linguistic construct to transform programs), and therefore special tool support (e.g., CorC [RSC$^+$19, BCK$^+$22]) is necessary to introduce the CbC refinement process to a programming language. As a result, the barrier to construct programs using CbC is large because the approach at first seems unintuitive and requires effort, knowledge, and special tool support.

In this article, we introduce two alternative correctness-by-construction development approaches that relax the inflexible CbC construction approach without losing the benefits of CbC itself. Both introduce more flexible language constructs to create programs which allow to condense construction steps that tackle the complex and strict programming style of CbC. The goal is to propose a usable CbC apporach that offers reasonable constructs to develop programs correctly. Therefore, we qualitatively discuss our two proposed approaches and the original CbC approach regarding their programming constructs, tool support, and usability to assess their benefits and drawbacks.

First, we present CbC-Block which adds new refinement rules. This introduction of new refinement rules should not be seen as a further restriction, but as a relaxation of the procedure. These new refinement rules increase the ways in which programs can be developed as they allow to refine abstract statements to a specified block of code that fulfills its specification. This basically means that this block can contain multiple assignments,

[1]The approach should not be confused with other CbC approaches such as CbyC of Hall and Chapman [HC02]. CbyC is a software development process that uses formal modeling techniques and analysis for various stages of development (architectural design, detailed design, code) to detect and eliminate defects as early as possible [Cha06]. We also exclude data refinement from abstract data types to concrete ones during code generation as for example in Isabelle/HOL [HKKN13].

selections, or loops whereas with classic CbC for each assignment, selection, and loop a new refinement step is needed. Initially, a block is just an abstract placeholder, but it has a pre-/postcondition specification so that the introduced specification of the block can be checked against the specification of the refined abstract statement. In a next step, the block is instantiated by some code, and it is directly proved that this code fulfills its own specification. The idea of the block rules is similar to a method call, but a block can alter local variables in the method under construction. A block of code can contain further blocks which can be subsequently refined. Consequently, any nesting of blocks may occur. CbC-Block is implemented as extension of the CorC tool support.

Second, we present TraitCbC which is a new software development approach that enables correct-by-construction development by method abstraction and method composition without relying on refinement rules and special tool support. TraitCbC uses *traits* [DNS+06], which are a flexible object-oriented language construct supporting a rich form of modular code reuse orthogonal to inheritance. A trait is a set of *concrete* or *abstract* methods (i.e., the method has either a body or has no body).[2] Traits can be composed into a larger trait or into a class that contains all methods of all composed traits. Trait composition exists as a direct concept of the programming language [DNS+06] instead of being a program transformation concept, such as the CbC refinement rules. On the basis of traits, TraitCbC introduces a programming guideline for an incremental program construction approach that guarantees that the resulting program is correct by construction. A construction step comprises the development of a method and direct composition with the existing code base to ensure correctness. TraitCbC allows the implementation of any method size and complexity as long as the methods are composable with respect to their specification. Even with this flexibility, TraitCbC keeps the advantages of a structured incremental development approach.

The contribution of our article is to demonstrate and compare the range of possibilities to develop programs correct by construction from strict rule-based CbC to the more relaxed CbC-Block to TraitCbC without any refinement rules. In this article, we introduce TraitCbC and explain the typing, reduction, and flattening rules. We give a proof that TraitCbC guarantees to develop programs correct by construction. We also present the CbC-Block approach with the block refinement rules. All approaches are implemented in the CorC [RSC+19] tool support. We compare and discuss classic CbC, CbC-Block, and TraitCbC qualitatively to assess their benefits and drawbacks. This article extends previous work [RPTS22] by introducing the typing and reduction rules of TraitCbC in detail. The soundness proof of TraitCbC is also presented in this article. CbC-Block is a new approach that has not been presented before.

2. Correctness-by-Construction

Classic correctness-by-construction (CbC) [Dij76, KW12, Mor94] is an incremental approach to construct programs. CbC uses a *Hoare triple* specification $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ stating that if the precondition `P` holds, and the statement `S` is executed, then the statement terminates and postcondition `Q` holds. The CbC refinement process starts with a Hoare triple where the statement `S` is abstract. This abstract statement can be seen as a hole in the program that needs to be filled. With a set of refinement rules, an abstract statement is replaced by more

[2] Java interfaces with default methods are a good approximation for what has been called trait in the literature

Definition 2.1: Refinement Rules for the Correctness-by-Construction Approach

Let P be the precondition, Q be the postcondition, and S be an abstract statement. Then, the Hoare triple $\{P\}S\{Q\}$ is **refinable** to

- **Skip:** $\{P\}\textbf{skip}\{Q\}$ *iff* P *implies* Q
- **Assignment:** $\{P\}\ \mathtt{x} := E\{Q\}$ *iff* P *implies* $Q[\mathtt{x} := E]$
- **Composition:** $\{P\}S_1; S_2\{Q\}$ *iff* intermediate condition M exists such that $\{P\}S_1\{M\}$ and $\{M\}S_2\{Q\}$ *hold*
- **Selection:** $\{P\}\textbf{if } G_1 \to S_1 \textbf{ elseif } \dots\ G_n \to S_n \textbf{ fi}\{Q\}$ *iff* P *implies* $G_1 \vee \dots \vee G_n$ and $\forall i \in \{1 \dots n\} : \{P \wedge G_i\}S_i\{Q\}$ *holds*
- **Repetition:** $\{P\}\textbf{do } [I, V]\ G \to S \textbf{ od}\{Q\}$ *iff* P *implies* I and $I \wedge \neg G$ *implies* Q and $\{I \wedge G\}S\{I\}$ *holds* and $\{I \wedge G \wedge V = V_0\}S\{I \wedge 0 \leq V < V_0\}$ *holds*
- **Weaken precondition:** $\{P'\}S\{Q\}$ *iff* P *implies* P'
- **Strengthen postcondition:** $\{P\}S\{Q'\}$ *iff* Q' *implies* Q
- **Method Call:** $\{P\}\mathtt{m}(a_1, \dots, a_n) \to b\{Q\}$ *iff* method $\{P'\}\mathtt{m}(p_1, \dots, p_n) \to r\{Q'\}$ exists and P *implies* $P'[p_i \setminus a_i]$ and $Q'[\text{old}(p_i) \setminus \text{old}(a_i),\ r \setminus b]$ *implies* Q

[RSC+19, KW12]

concrete statements (i.e., statements in the guarded command language [Dij76] that can contain further abstract statements). The process stops, when all abstract statements are refined to concrete statements so that no holes remain in the program. As each refinement rule is sound and each correct application of a refinement rule guarantees to satisfy the starting Hoare triple, the resulting program is correct by construction [KW12]. The CbC approach is strictly tied to this set of predefined refinement rules. A developer cannot deviate from this concept.

In Definition 2.1, we present the eight refinement rules of CbC by Kourie and Watson [KW12]. The concrete program statements are written in the guarded command language [Dij75]. To apply a refinement rule, it has to be checked that side conditions of the rule application are satisfied. This is done by pen-and-paper or with specialized tools [RSC+19]. For example, the skip rule introduces an empty statement that does not alter the program state. This refinement is applicable if and only if the precondition P implies the postcondition Q. The composition rule splits the Hoare triple $\{P\}S\{Q\}$ into two Hoare triples by using an intermediate condition M. This refinement is applicable, if and only if the two new Hoare triples are correct. Of course, the statements S_1 and S_2 are still abstract and can be further refined.

3. CbC-Block— CbC With Block Contracts

In this section, we introduce the CbC-Block approach that adds two new refinement rules to classic CbC. The new refinement rules increase the ways to construct programs. Therefore, the rigid CbC approach is loosened while retaining the benefits of a structured program construction approach. A block rule refines an abstract statement to a block that is specified with a block contract (i.e., a pre-/postcondition specification for that block) [ABB+16]. The block is a special statement that can be further refined in two ways. Similar to an abstract statement, any CbC refinement rules can be applied. Additionally, the block can be instantiated by any sequence of concrete statements and further blocks with a block-instantiation rule. Thus, a block can be used to condense the application of several CbC refinement rules. For example, a block can be instantiated with a while-loop that already contains a concrete body. This instantiation replaces the application of the repetition rule and at least one assignment rule. We introduce CbC-Block with a motivating example and present the block rules to introduce and instantiate blocks. In the end of this section, we present CbC-Block implemented in the CbC tool CorC[3] and evaluate its usability with a user study.

3.1. **Motivating Example.** In this section, we exemplify the CbC-Block approach by implementing a `maxElement` algorithm. The `maxElement` algorithm searches the largest element in a list of integers. The list supports a `get`-method which returns the element at the specified position in this list. A `contains`-method checks that the result is a member of the list. We iterate with a while-loop through the list and use local variables to temporally save the current largest element. We use Java and JML [LBR98] as programming and specification language in the example.

In Listing 1, we start implementing method `maxElement` that is specified with a pre- and postcondition contract. The precondition states that the list must contain at least one element. The postcondition states that the largest element in the list is returned. In this example, we start with a program where some CbC refinement rules are already applied, and then, apply the block rules to finish the implementation.

The program is already split into three parts using the composition refinement rule with two intermediate conditions between them. In the first part, two local variables `i` and `j` are introduced with the assignment rule. The variable `i` is used to temporally store the largest element. In the beginning, the largest element (up to that point) is the first element in the list. The variable `j` is our loop variable to iterate through the list. In the third program part, variable `i` is returned. We start with this program state to show that CbC-Block also supports the standard CbC approach, but we will now use the block rules to exemplify the benefits of CbC-Block.

In the second part between the two intermediate conditions, the block rule of CbC-Block is applied. The block `B1` is specified with a block contract in lines 13–16. For the functional behavior, we specify the values of `i` and `j`, and the size of the list in the precondition. This specification is equal to the preceding intermediate condition. The postcondition of the block states that `i` is the largest element. This postcondition meets the intermediate condition after the block. Therefore, we know that the program is correct under the assumption that block `B1` fulfills its specification. In the next steps, we concretize the block, and the applied refinement rule guarantees that the instantiated block is correct according to its specification. We can concretize the block either in one step, by instantiating the block with concrete Java code or stepwise by instantiating the block with some Java statements and other blocks.

We decide to partially implement the block. In Listing 2, we define the block that should be refined by referring to block `B1` in line 1 and repeat the specification of that block. Inside the curly brackets the instantiation is shown. We implement the block with a while-loop. We iterate through the list as long as variable `j` is smaller than the size of the list. This is stated in the loop guard. The loop is specified with a loop invariant in lines 10–12. Thereby,

[3]`https://github.com/KIT-TVA/CorC`

```
/*@ public normal_behavior
  @ requires list.size() > 0;
  @ ensures list.contains(\result) &&
  @   (\forall int q; q >= 0 && q < list.size(); \result >= list.get(q));
  @*/
  public int maxElement(List list) {
    int i = list.get(0);
    int j = 1;

  //@ Intm: list.size() > 0 && i == list.get(0) && j == 1;

  /*@
    @ normal_behavior
    @ requires list.size() > 0 && i == list.get(0) && j == 1;
    @ ensures list.contains(i) &&
    @   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));
    @*/
    { \Block B1; }

  //@ Intm: list.contains(i) &&
  //@   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));

    return i;
  }
```

Listing 1: Initial program of `maxElement`

the variable `i` stores the largest element of the already checked elements up to the index `j`. The index `j` is inside the bounds of the list. We use the difference between the size of the list and `j` as loop variant. As variable `j` increases in each iteration, the difference decreases, and the loop thus terminates. The increase of `j` is already implemented at the end of the while-loop. The body of the loop contains another block `B2` in lines 15–22. The precondition of the block is the loop invariant with the difference that we know that variable `j` is smaller than the size of the list. This block should update variable `i` that contains the largest element. We want to compare the largest element with the next element in the list. If that element is larger, variable `i` is updated. We checked one more element of the list, and therefore, we increase the range of the universal quantifier in the postcondition. This instantiation condenses the application of three CbC refinement rules, the repetition rule to create the loop, a composition rule, and an assignment rule for the loop body.

The next step is to verify that the instantiation satisfies the block contract. Starting with the precondition and after executing the introduced instantiation, the postcondition of the block contract must be fulfilled. The details of checking this instantiation are explained in the next section. When the correctness of this instantiation is shown, we can continue to instantiate the next block `B2`.

In Listing 3, the instantiation of block `B2` implements the case when a larger element is found. The functional pre- and postcondition of the block differ by the range of considered elements. In the postcondition, the range is increased by one. In this block, we compare the current largest element `i` with the element at index `j`. If the element at index `j` is larger, we

```
Block B1;

/*@
  @ normal_behavior
  @ requires list.size() > 0;
  @ ensures list.contains(i) &&
  @   (\forall int q; q >= 0 && q < list.size(); i >= list.get(q));
  @*/
  {
  //@ loop_invariant list.contains(i) && j > 0 && j <= list.size() &&
  //@   (\forall int q; q >= 0 && q < j; i >= list.get(q));
  //@ decreases list.size() - j;
    while (j < list.size()) {

    /*@
      @ normal_behavior
      @ requires list.contains(i) && j > 0 && j < list.size() &&
      @   (\forall int q; q >= 0 && q < j; i >= list.get(q));
      @ ensures list.contains(i) && j > 0 && j < list.size() &&
      @   (\forall int q; q >= 0 && q < j+1; i >= list.get(q));
      @*/
      { \Block B2; }

      j = j + 1;
    }
  }
```

Listing 2: Refinement of block B1

update variable i. In the other case, i is still the largest element and not updated. Again, we condense CbC refinement rules by instantiating the block with concrete code. We have to verify that the instantiation is correct. If this is done, we have finished the refinement process because no further block or any abstract statement is left.

By guaranteeing the correctness of all refinement steps, we can conclude that the whole program is correct by construction. The resulting program is shown in Listing 4. Here, the blocks are recursively replaced with their instantiation. The specification is limited to the method contract and the loop invariant and variant annotations. By stepwise refining the program, we can detect errors when proving single refinement steps. This locality of information helps to track down errors more easily than with monolithic post-hoc verification.

3.2. **Block Refinement Rules of CbC-Block.**

In this section, we describe how refinement rules are added to establish the CbC-Block approach. We describe the refinement rule to introduce a block and the refinement rule to instantiate a block with concrete code.

For the block rules to be syntactically applicable, we extend Java to write a block with a name. Normally, a block in Java is just a sequence of Java statements inside curly brackets. In addition, Ahrendt et al.[ABB$^+$16] defined block contracts to specify the behavior of a Java block similar to a method [Mey92, Lei95]. To establish a CbC refinement process, we introduce a specified block as an abstract statement in CbC-Block with an according

```
Block B2;

/*@
  @ normal_behavior
  @ requires list.contains(i) && j > 0 && j < list.size() &&
  @   (\forall int q; q >= 0 && q < j; i >= list.get(q));
  @ ensures list.contains(i) && j > 0 && j < list.size() &&
  @   (\forall int q; q >= 0 && q < j+1; i >= list.get(q));
  @*/
  {
    if (list.get(j) > i) {
      i = list.get(j);
    }
  }
```

Listing 3: Refinement of block `B2`

```
/*@ public normal_behavior
  @ requires requires list.size() > 0;
  @ ensures list.contains(\result) &&
  @   (\forall int q; q >= 0 && q < list.size(); \result >= list.get(q));
  @*/
  public int maxElement(List list) {
    int i = list.get(0);
    int j = 1;
  //@ loop_invariant list.contains(i) && j > 0 && j <= list.size() &&
  //@   (\forall int q; q >= 0 && q < j; i >= list.get(q));
  //@ decreases list.size() - j;
    while (j < list.size()) {
      if (list.get(j) > i) {
        i = list.get(j);
      }
      j = j + 1;
    }
    return i;
  }
```

Listing 4: Final implementation of `maxElement`

refinement rule. In the refinement rule, we use the Hoare triple notation that is also used for the classic CbC refinement rules. We focus on functional pre-/postconditions and exclude regular and irregular termination of blocks for CbC-Block. For the instantiation of a block (e.g., to write a sequence of Java statements that fulfill the specification), we follow the syntax of a concrete block in Java, but we add a name for reference.

An abstract statement is refined by the block-introduction rule to a block with a name and a block contract. Thus, a block name is an abstract placeholder. The side condition of the refinement rule guarantees the correctness of the program to be developed. For the block-introduction rule, we have to check three parts. First, the precondition of the refined abstract statement must imply the precondition of the block. This ensures that the pre-state of the block is satisfied, and the block can be executed. Second, the postcondition of the block must imply the postcondition of the refined abstract statement to continue the program after the block. Third, the block must satisfy its own contract. As the block can be seen as a Hoare triple, any CbC refinement rule can be applied to the block. The check of the side condition of the applied refinement rule guarantees the correctness of the block under development.

Rule 3.1 (Block-Introduction)**.** *Hoare triple* $\{P\}S\{Q\}$ *is* ***refinable*** *to* $\{P'\}$ *`Block B`* $\{Q'\}$ iff P implies P' *and* Q' implies Q *and* $\{P'\}$ *`Block B`* $\{Q'\}$ holds*.*

With the block-instantiation rule, we allow to instantiate a block with concrete code that can contain further blocks (see the instantiation in Listing 2). For application, it must be checked that this instantiation fulfills the block contract. We use the capabilities of program verification. We translate the block to a method and verify whether this translated *block-method* fulfills its contract. Thus, we have to prove that the dynamic formula $P \rightarrow$ `<statement;...>`Q is fulfilled. Assuming the precondition, the postcondition must be satisfied after executing the statements in the block. Dynamic logic extends first-order logic with two operators. A diamond modality `<p>`Q and a box modality `[p]`Q with a program `p` and a dynamic logic formula Q. Intuitively, the diamond modality states total correctness of the program, and partial correctness is stated with the box modality.

The translation from a block to a block-method is as follows. The block contract is translated to the contract of the block-method. The translated block-method is added to the same class as the method in which the block is declared. The statements within the block become the body of the block-method. As block could introduce local variables that are already declared in the surrounding method [KFFD86], an α-conversion [Bar84] is necessary to safely rename identifiers. A block does not have the same scope of a complete method and neither has parameters nor a return type. Declarations of parameters and local variables have to be added to the block-method, so that is has the same scope as the method. Therefore, we translate accessible variables of the block to parameters of the block-method, and assignable variables of the block to fields of the class containing the block-method. This differentiation is done because a contract can only access parameter values before execution of the method, but it can access the modified values of fields. Accessible or assignable fields of the class are usable because the block-method is added to the class for verification purposes. The return type of the block-method is `void` because we exclude the use of return statements inside the block. This transformation is limited in its expressiveness as we are excluding irregular termination, but sufficient to demonstrate the correctness-by-construction process for normal execution.

Rule 3.2 (Block-Instantiation)**.** *Hoare triple* $\{P\}$ *`Block B`* $\{Q\}$ *is* ***refinable*** *to* $\{P\}$ *`<statement;...>`* $\{Q\}$ iff $P \rightarrow$ *`<statement;...>`* Q*, where `<statement;...>` is any sequence of concrete program statements possibly containing further blocks.*

3.3. Discussion.

In this subsection, we discuss the block refinement rules in comparison to related approaches that allow to introduce code sequences, such as method calls, macro expansions, and abstract execution.

Difference to the Method Call Rule. The difference between the block refinement rules and the method call refinement rule is that for a method call only the contract is used to verify correctness of the caller. The content of the method is assumed to be correct with respect to the method's contract. With the block rules, both the contract and the content of the block are always checked for correctness. A big difference between the block rules and the method call rule is their scope. In a method, only variables of the method are changed and no local variables of the calling context. A block allows the modification of local variables as demonstrated in the motivating example.

Difference to Macro Expansion. Macro expansion is a textual transformation of input source code. A preprocessor replaces macros with concrete source code. This is similar to the block-instantiation rule, where a block name is replaced with concrete source code. As for our block-instantiation rule, a macro expansion can capture identifiers already used in the surrounding scope. Therefore, hygienic macro expansion uses α-conversion [Bar84] to rename identifiers. The difference to CbC-Block is that CbC-Block demands a specification for a block that is introduced in the block-introduction rule. Additionally, the block-instantiation rule starts a procedure that verifies whether the block instantiation fulfills its specification. A macro expansion is just a transformation of code.

Abstract Execution for Correctness-by-Construction. Abstract execution [SH19] is a technique to specify and verify programs with partially abstract parts. Abstract execution generalizes symbolic execution. It is tailored to Java, but the principles are applicable to other sequential languages. Java and JML are extended with the concept of abstract program element (APE); an abstract statement or an abstract expression. An APE is a placeholder for any program part with or without side effects. To verify the correctness of programs containing abstract program elements, these elements are specified with a contract similar to a block contract. The extended specification language of abstract execution allows to specify the behavior of the program element in cases of regular or irregular termination including side effects [SH19]. The strength of abstract execution is the reasoning of irregular termination that we exclude in CbC-Block.

APEs can be used similar to blocks of CbC-Block to establish a process for refinement-based program construction. With abstract execution, we write programs containing APEs. These programs can be verified to be correct under the assumption that the APEs fulfill their specifications. In a refinement step, an APE is replaced by a program part that contains concrete statements and possibly other abstract program elements. We have to verify that the insertion fulfills the specification of the refined APE. This refinement is repeated until no APE remains. Similar to classic CbC, this process does not require a program to be monolithically verified, but it is sufficient that each APE replacement is verified to conclude that the program is correct by construction. This process is the same as for CbC-Block if we always instantiate a block without using any other CbC refinement rule. We still argue that the application of other CbC refinement in tandem with blocks is beneficial because they enforce a structured program construction process where developers think about the implementation more thoroughly. Therefore, we decided for CbC-Block as presented instead of utilizing abstract execution because CbC-Block is the sweet spot between expressiveness and changes to the program construction process of classic CbC. Combining classic CbC refinement rules with abstract execution requires major changes to classic CbC, so that the strength of abstract execution is usable (i.e., the refinement rules must be adapted to consider irregular termination).

3.4. **Implementation.** In this subsection, we describe the implemented tool support for CbC-Block. Classic CbC is already supported by the CorC tool [RSC+19]. CorC has a graphical and a textual editor to develop programs. In this work, we extend CorC with the new rules of CbC-Block. The textual IDE is implemented in Eclipse with Xtext.[4] Xtext provides the functionality to develop IDEs for domain-specific languages. We use Xtext to establish an editor for CorC-programs that consist of JML, Java, and the CbC-specific keywords. The grammar of a CorC-program, which represents a refinement-based construction according to CbC, is defined in Xtext. Based on this grammar, Xtext supports syntax checks, highlighting and auto completion.

In CorC, we implemented the refinement rules of Definition 2.1. For CbC-Block, we added refinement rules of block-introduction and block-instantiation. An instantiation is written in a separated program starting with the name of the refined block and followed by the contract and the block's implementation. To verify that a block-instantiation fulfills its block contract, a generator is implemented. It transforms a block-instantiation to a method and starts the verification process by calling KeY [ABB+16]. The transformation to a method follows the concept presented for the block-instantiation rule before. The generator also creates the final method implementation if all refinement steps are proven. All block instantiations must be recursively inserted to get the final method implementation. The final method implementation can be integrated into an existing code base. This generator can also construct partial methods when some parts are not fully refined. This is helpful in intermediate steps of the construction to retain an overview of the current method.

3.5. **Evaluation with a User Study.** We evaluate CbC-Block with a user study. We compare CbC-Block with classic CbC by (dis)allowing the use of the block rules to answer the following research question.

RQ1: Does the CbC-Block approach improve classic CbC in terms of usability?

In the user study, we engaged five participants that know classic CbC and the CorC tool. Knowledge of classic CbC and CorC is a necessary prerequisite because a new feature was evaluated that could only be understood if the participants already knew the classic CbC approach in CorC. Then, they can estimate the benefits of the new block refinement rules. With this small number of participants, it was possible that everyone could solve the study tasks consecutively.

Each participant had to implement two algorithms, one algorithm with CbC-Block and the block rules and one algorithm with classic CbC and without the block rules. The algorithms are `maxElement` and `dutchFlag`. The `maxElement` algorithm was already introduced as the motivating example. The `dutchFlag` algorithm sorts a list containing only three different elements. In the original description, each element has either a red, a blue or a white color. The elements of the list are to be reordered so that the list results in the national flag of the Netherlands (red, white, blue). We adapted the task to a list that contains an unknown quantity of the numbers 0, 1, and 2. Both algorithms can be implemented in a few lines of code with one loop through the list of elements. As both algorithms are explained to the participants, we expected that the correct implementation is possible without major

[4] `https://www.eclipse.org/Xtext/`

problems. For each task, they had 30 minutes. We split the participants in two groups using the Latin Square design [WRH+12]. Each group implemented the algorithms in the same order, but the approaches were used crosswise to address possible learning effects through an order of the tools. After implementing both algorithms, we conducted a structured interview. The questions of the interview are presented in Appendix A.

We summarize the most important answers of the participants and discuss the findings. The tasks were correctly solved by all participants. In general, the participants needed more time for the task with CbC-Block than for the task with classic CbC. As we only had five participants, these statistics are only of limited significance. The participants followed the CbC-Block approach and refined a program stepwise using the block rule. All participants considered the introduction of the block rules to be useful. The rules save the application of other CbC refinement rules during construction. For CbC-Block, the participants positively mentioned the familiarity with a textual editor, the grouping of statements for one refinement step by using a block, and the freedom to not be bound to the classic CbC refinement rules. For classic CbC, the answers are in line with previous user studies [RTC+19, RBTS21]. The participants positively mentioned the visual overview of the refinements and the status of the verification. They liked the fine-grained feedback for every applied refinement rule. As CbC-Block extends CbC, these positive answers also apply for CbC-Block. The participants stated for both approaches that the incremental construction helps to track down errors. The participants still miss more assistance if a proof cannot be closed.

While we observed the participants, we noticed that both approaches need a correct and sufficient specification as a starting point. If that is the case, refining and checking side-conditions can be very successful. If, on the contrary, the specification needs to be adjusted in the process, the effort to verify the program increases drastically. With classic CbC, the participants are forced into the process of refining and verifying top-down. With CbC-Block, the participants have more freedom to develop the program.

Regarding the finished tasks, all participants had experience with CorC. Therefore, it is not surprising that all participants finished the task. They never used CbC-Block before, but the participants conceptually understood the features of CbC-Block and accepted the expansion well. Nonetheless, the participants need more time to fully understand the IDE and the programming workflow of CbC-Block. This results in a longer time to implement the algorithms.

We can answer **RQ1** that CbC-Block is a promising feature to increase the usability of CorC, but as for each new feature, developers need time to get used to. Some answers of the participants highlight that with more training and better tool support, they are willing to use the CbC-Block approach to construct correctness-critical programs.

Threats to Validity. In our user study, we had only five participants. Due to the small number of participants, the qualitative results that we collected in the structured interview are not generalizable. Nevertheless, the results are relevant, since the users are experts in CorC and can therefore assess the advantages of the extension. A more comprehensive evaluation with non-experts is not possible because they cannot properly interact with the tool. The participants only implemented and verified two small algorithms in our experiment, and therefore, we cannot generalize the results to larger problems.

4. TraitCbC

In this section, we introduce TraitCbC with a motivational example. We present TraitCbC formally and prove soundness of the TratiCbC program construction approach. In the end of this section, we show the proof-of-concept implementation and a feasibility evaluation.

TraitCbC uses method abstraction and method composition to enable an incremental CbC-based development approach. This approach on method level allows a flexible way to construct the desired program with any number and size of auxiliary methods. A developer starts by implementing a method (e.g., a method `a`) in a first trait. Similar to classic CbC, the method can contain holes that are refined in subsequent steps. A hole in TraitCbC is an abstract method (e.g., an abstract method `b`) that is called in method `a`; that is, a call to an abstract method corresponds to an abstract statement in classic CbC. In the next step, one of these new abstract methods (e.g., `b`) is implemented in a second trait, again more abstract methods can be declared for the implementation. To be correct, it must be proven that each implemented method satisfies its specifications. Afterwards, the traits are composed; the composition operation checks that the specification of the concrete method `b` in the second trait fulfills the specification of the abstract method `b` in the first trait. This incremental program construction approach stops when the last abstract method is implemented, and all traits are composed.

4.1. Motivating Example.

We illustrate using an example of how TraitCbC enables CbC using traits. We use an object-oriented language in the code examples. In Listing 5, we construct a method `maxElement` that finds the maximum element in a list of numbers. We slightly adjust the implementation of the algorithm to better fit for TraitCbC. With TraitCbC, we have an abstraction on method level. We utilize methods to outsource program pieces that can be reused (i.e., we want to implement methods that are verified once, but called several times in a program to reduce verification effort).

In this `maxElement` example, a list has a head and a tail. Only non-empty lists have a maximum element. This is explicit in the precondition of our specification, where we require that the list has at least one element. In the postcondition, we specify that the result is in the list and larger than or equal to every other element. In the first step, we create a trait `MaxETrait1` that defines the abstract method `maxElement`. The method `maxElement` is abstract, i.e., equivalent to an abstract statement in CbC.

```
trait MaxETrait1 {
  @Pre: list.size() > 0
  @Post: list.contains(result) &
    (forall Num n: list.contains(n) ==> result >= n)
  abstract Num maxElement(List list);
}
```

Listing 5: Initial trait for maxElement

In the second step in trait `MaxETrait2` in Listing 6, we implement the method `maxElement` using two abstract methods. We introduce an `if`-`elseif`-`else`-expression where the branches invoke abstract methods. The guards check whether the list has only one element or whether the current element is larger than or equal to the maximum of the rest of the list. The abstract method `accessHead` returns the current element, and the abstract method `maxTail` returns the maximum in the remaining list. So, we recursively search the list for the largest element by comparing the maximum element of the list tail with the current element until we reach the end of the list.

```
trait MaxETrait2 {
  @Pre: list.size() > 0
  @Post: list.contains(result) &
    (forall Num n: list.contains(n) ==> result >= n)
  Num maxElement(List list) =
    if (list.size() == 1) {accessHead(list)}
    elseif (accessHead(list) >= maxTail(list))
      {accessHead(list)}
    else {maxTail(list)}

  @Pre: list.size() > 0
  @Post: result == list.element()
  abstract Num accessHead(List list);

  @Pre: list.size() > 1
  @Post: list.tail().contains(result) &
    (forall Num n: list.tail().contains(n) ==> result >= n)
  abstract Num maxTail(List list);
}
```

Listing 6: Implementation of maxElement with auxiliary methods

The correct implementation of the method `maxElement` can be guaranteed under the assumptions that all introduced abstract methods are correctly implemented. Similar to post-hoc verification, a program verifier conducts a proof of method `maxElement` and uses the introduced specifications of the methods `accessHead` and `maxTail`. If the proof succeeds, we know that the first method is correctly implemented. In our incremental CbCTrait approach, we verify each method implementation directly after construction; and so we are able to reuse each implemented method in the following steps (e.g., by calling the method in the body of other methods).

We now compose the developed traits to complete the first construction step. To perform the composition `MaxETrait1` + `MaxETrait2`, we check that the specification of the method `maxElement` fulfills the specification of the abstract method in the first trait (cf. Liskov substitution principle [LW94]). In this case, this means checking that:
`MaxETrait1.maxElement(..).pre ==> MaxETrait2.maxElement(..).pre` as well as:
`MaxETrait2.maxElement(..).post ==> MaxETrait1.maxElement(..).post.`
When the composition of two verified traits is successful, the result is also a verified trait. Note that the composed trait does not need to be verified directly by a program verifier in TraitCbC because it is correct by construction. In this example, the specifications are the same, thus checking for a successful composition is trivial, but this is not generally the case. In particular, the logic needs to take into account ill-founded specifications and recursion in the specification. We discuss more about the difficulties of handling those cases in previous work [RPTS22].

The methods `accessHead` and `maxTail` are implemented in the next two construction steps in traits `MaxETrait3` and `MaxETrait4`[5]. The implementations are shown in Listing 7 and in Listing 8. As we implement a recursive method, the method `maxTail` calls the `maxElement` method, thus `maxElement` is introduced as an abstract method in this trait. We have to verify that the method `accessHead` satisfies its specification using a program verifier. Similarly, we have to verify the correctness of the method `maxTail`.

```
trait MaxETrait3 {
  @Pre: list.size() > 0
  @Post: result == list.element()
  Num accessHead(List list) = list.element()
}
```

Listing 7: Implementation of accessHead

```
trait MaxETrait4 {
  @Pre: list.size() > 1
  @Post: list.tail().contains(result) &
    (forall Num n: list.tail().contains(n) ==> result >= n)
  Num maxTail(List list) = maxElement(list.tail())

  @Pre: list.size() > 0
  @Post: list.contains(result) &
    (forall Num n: list.contains(n) ==> result >= n)
  abstract Num maxElement(List list);
}
```

Listing 8: Implementation of maxTail

As before, all traits are composed, and it is checked that the specifications of the concrete methods fulfill the specifications of the abstract ones. As we have no contradicting specifications for the same methods, the composition is well-formed. In Listing 9, the final program `MaxE` is shown. All traits are composed.

```
class MaxE = MaxETrait1 + MaxETrait2 + MaxETrait3 + MaxETrait4
```

Listing 9: Trait composition

The already proven auxiliary methods in traits can be reused. For example, if we want to implement a `minElement` method as shown in Listing 10, we could reuse already implemented traits to reduce the programming and verification effort. The method `minElement` is implemented in the following in trait `MinE` with one abstract method. The specification of the method `accessHead` is the same as for the method `accessHead` above, so `MaxETrait3` can be reused. In this example, we show the flexible granularity of TraitCbC by directly implementing the else branch, instead of introducing an auxiliary method as for `maxElement`.

[5]The methods could also be implemented in one trait.

```
trait MinE {
  @Pre: list.size() > 0
  @Post: list.contains(result) &
    (forall Num n: list.contains(n) ==> result <= n)
  Num minElement(List list) =
    if (list.size() == 1) {accessHead(list)}
    elseif (accessHead(list) <= minElement(list.tail()))
      {accessHead(list)}
    else {minElement(list.tail())}

  @Pre: list.size() > 0
  @Post: result == list.element()
  abstract Num accessHead(List list);
}
```

Listing 10: Implementation of minElement with auxiliary method accessHead

The correctness of `minElement` is verified with the specifications of the method `accessHead`. By composing `MinE` with `MaxETrait3`, we get a correct implementation of `minElement`. Note how this verification process supports abstraction: as long as the contracts are compatible, methods can be implemented in different styles by different developers to best meet non-functional requirements while preserving the specified observable behavior [tBCSW18]. A completely different implementation of `maxElement` can be used if it fulfills the specification of the abstract method `maxElement` in trait `MaxETrait1`. This decoupling of specification and corresponding satisfying implementations facilitates an incremental program construction approach where a specified code base is extended with suitable implementations [DDJS14].

4.2. **Object-Oriented Trait-Based Language.** In this section, we formally introduce the syntax, type system, reduction, and flattening semantics of a minimal core calculus for TraitCbC. We keep this calculus for TraitCbC parametric in the specification logic so that it can be used with a suitable program verifier and associated logic. The presented rules to compose traits are conventional. The focus of our work is to enable a CbC approach using traits that developers can easily adopt. Therefore, we present the calculus to prove soundness of TraitCbC, but focus on the presentation of the advantages of incremental trait-based programming in this paper. Indeed, languages with traits and with a suitable specification language intrinsically enable incremental program construction.

4.2.1. *Syntax.* The concrete syntax of our core calculus for TraitCbC is shown in Fig. 1, where non-terminals ending with 's' are implicitly defined as a sequence of non-terminals, i.e., $vs ::= v_1 \dots v_n$. We use the metavariables t for trait names, C for class names and m for method names. A program consists of trait and class definitions. Each definition has a name and a trait expression E. The trait expression can be a *Body*, a trait name, a composition of two trait expressions E, or a trait expression E where a method is made abstract, written as E[`makeAbstract` m]. A *Body* has a flag `interface` to define an interface, a set of implemented interfaces Cs and a list of methods Ms. Methods have a method header MH consisting of a specification S, the return type, a method name, and a list of parameters. Methods have an optional method body. In the method body, we have

$$
\begin{array}{lll}
\mathit{Prog} & ::= & \mathit{Ds}\ e \\
D & ::= & \mathit{TD} \mid \mathit{CD} \\
\mathit{Name} & ::= & t \mid C \\
\mathit{TD} & ::= & t = E \\
\mathit{CD} & ::= & C = E \\
E & ::= & \mathit{Body} \mid t \mid E + E \mid E[\texttt{makeAbstract}\ m] \\
\mathit{Body} & ::= & \{\texttt{interface}?\ [\mathit{Cs}]\ \mathit{Ms}\} \\
M & ::= & \mathit{MH}\ e?; \\
\mathit{MH} & ::= & S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n) \\
e & ::= & x \mid e.m(es) \mid \texttt{new}\ C(es) \\
\mathcal{E}_v & ::= & [].m(es) \mid v.m(vs\ []\ es) \mid \texttt{new}\ C(vs\ []\ es) \\
v & ::= & \texttt{new}\ C(vs) \\
\Gamma & ::= & x_1 : C_1 \ldots x_n : C_n \\
S & ::= & \ldots \textit{e.g.}\ \texttt{Pre} : P\ \texttt{Post} : P \\
P & ::= & \ldots \textit{e.g.}\ \text{First order logic}
\end{array}
$$

Figure 1: Syntax of the trait system

standard expressions, such as variable references, method calls, and object initializations. For simplicity, we exclude updatable state. Field declarations are emulated by method declarations, and field accesses are emulated by method calls.

The specification S in each method header is used to verify that methods are correctly implemented. The specification is written in some logic. In our examples, we will use first-order logic (cf. the example in Section 4.1). A well-formed program respects the following conditions:

Every *Name* in *Ds* must be unique so that *Ds* can be seen as a map from names to trait expressions. Trait expressions E can refer to trait names t. A well-formed *Ds* does not have any circular trait definitions like $t = t$ or $t_1 = t_2$ and $t_2 = t_1$. In a *Body*, all names of implemented interfaces must be unique and all method names must be unique, so that *Body* is a map from method names to method definitions. In a method header, parameters must have unique names, and no explicit parameter can be called `this`.

4.2.2. *Typing Rules.* In our type system, we have a typing context $\Gamma ::= x_1 : C_1 \ldots x_n : C_n$ which assigns types C_i to variables x_i. We define typing rules for our three kinds of expressions: x, method calls, and object initialization. We combine typing and verification in our type checking $\Gamma \vdash e : C \dashv P_0 \models P_1$. This judgment can be read as: under typing context Γ, the expression e has type C, where under the knowledge P_0 we need to prove P_1. The knowledge P_0 is our collected information that we use to prove a method correct. That means, in our typing rules, we collect the knowledge about the parameters and expressions in a method body to verify that this method body fulfills the specification defined in the method header. The verification obligation P_1 should follow from the knowledge P_0.

We check if methods are well-typed with judgments of form $\mathit{Ds}; \mathit{Name} \vdash M : \mathit{OK}$. This judgment can be read as: in the definition table, the method M defined under the definition *Name* is correct. The typing rules of Fig. 2 are explained in the following. The first four rules type different expressions and collect the information of these expressions to prove with rule MOK that a method fulfills its specification. In the rule MOK with keyword **verify**,

$$\frac{}{\Gamma \vdash x : \Gamma(x) \dashv \texttt{result} : \Gamma(x) \ \& \ \texttt{result} = x \models \mathit{true}} \ (\textsc{x})$$

$$\frac{\begin{array}{c} S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n)\_; \in \ \mathit{methods}(C_0) \qquad \Gamma \vdash e_0 : C_0 \dashv P_0 \models P_0' \ \ldots\ \Gamma \vdash e_n : C_n \dashv P_n \models P_n' \\ x_0' \ldots x_n'\ \mathit{fresh} \qquad S' = S[\texttt{this} := x_0',\ x_1 := x_1',\ \ldots,\ x_n := x_n'] \\ P = (\texttt{result} : C \ \&\ P_0[\texttt{result} := x_0'] \ \&\ \ldots\ \&\ P_n[\texttt{result} := x_n'] \&\ (\mathit{Pre}(S') \implies \mathit{Post}(S'))) \end{array}}{\Gamma \vdash e_0.m(e_1 \ldots e_n) : C \dashv P \models P_0' \ \&\ \ldots\ \&\ P_n' \ \&\ \mathit{Pre}(S')} \ (\textsc{Method})$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : C_1 \dashv P_1 \models P_1' \ \ldots\ \Gamma \vdash e_n : C_n \dashv P_n \models P_n' \\ \mathit{getters}(C) = S_1\ \texttt{method}\ C_1\ x_1();\ \ldots\ S_n\ \texttt{method}\ C_n\ x_n(); \qquad x_1' \ldots x_n'\ \mathit{fresh} \qquad S_i' = S_i[\texttt{this} := \texttt{result}] \\ P_i'' = (P_i[\texttt{result} := x_i'] \ \&\ (\mathit{Pre}(S_i') \implies \texttt{result}.x_i() = x_i')) \qquad P = (\texttt{result} : C \ \&\ P_1'' \ \&\ \ldots\ \&\ P_n'') \end{array}}{\Gamma \vdash \texttt{new}\ C(e_1 \ldots e_n) : C \dashv P \models P_1' \ \&\ \ldots\ \&\ P_n' \ \&\ \mathit{Pre}(S_1') \ \&\ \ldots \&\ \mathit{Pre}(S_n')} \ (\textsc{New})$$

$$\frac{\Gamma \vdash e : C' \dashv P \models P' \qquad C'\ \mathit{instanceof}\ C}{\Gamma \vdash e : C \dashv P \models P'} \ (\textsc{sub})$$

$$\frac{\begin{array}{c} \Gamma = \texttt{this} : \mathit{Name},\ x_1 : C_1,\ \ldots,\ x_n : C_n \qquad \Gamma \vdash e : C \dashv P \models P' \\ \textbf{verify}\ \mathit{Ds} \vdash (\Gamma \ \&\ \mathit{Pre}(S) \ \&\ P) \models (P' \ \&\ \mathit{Post}(S)) \end{array}}{\mathit{Ds};\ \mathit{Name} \vdash S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n)\ e;\ : \mathit{OK}} \ (\text{MOK})$$

$$\frac{}{\mathit{Ds};\ \mathit{Name} \vdash S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n);\ : \mathit{OK}} \ (\textsc{AbsMOK})$$

$$\frac{\begin{array}{c} \mathit{Body} = \{\texttt{interface}?\ [\mathit{Cs}]\ M_1 \ldots M_n\} \\ \mathit{Ds}; \mathit{Name} \vdash M_1 : \mathit{OK} \ldots \mathit{Ds}; \mathit{Name} \vdash M_n : \mathit{OK} \end{array}}{\mathit{Ds}; \mathit{Name} \vdash \mathit{Body}\ : \mathit{OK}} \ (\textsc{BodyTyped})$$

Figure 2: Expression typing rules of TraitCbC

we call a verifier to prove each method once. Abstract methods (AbsOK) are always correct. Rule BodyTyped ensures that all methods in a body are correctly typed.

x : As usual, the type of a variable is stored in the environment Γ. From the verification perspective, we do not need to prove anything to be allowed to use a variable; thus we use *true*. We know that the result of evaluating a variable is the value of such variable, and that such value is of the type of the variable; thus we have $\texttt{result} : \Gamma(x)\ \&\ \texttt{result} = x$. The `result` is the returned value of evaluating this expression, and *variable* : *type* is a predicate in our system. As you can notice, we are assuming that our parametric logic supports at least a logical *and* (&); but of course other ways to merge knowledge could work too.

Method: As usual, to type a method call, we inductively type the receiver and all the parameters. In this way, we obtain all the types $C_0 \ldots C_n$, all the knowledge $P_0 \ldots P_n$, and all the verification obligations $P_0' \ldots P_n'$. Inside of all conditions $P_i \models P_i'$ we call the result of e_i `result`. We cannot simply merge the knowledge of $P_0 \ldots P_n$, since their `result` refers to different concepts. Thus, we chose fresh $x_0' \ldots x_n'$ variables, and we rename `result` of P_i and P_i' into x_i'. Similarly, S' is the specification of the method adapted using $x_0' \ldots x_n'$.

The verification obligation of course contains all the obligations of the receiver and the parameters, but also requires the precondition of the method to hold.

The knowledge contains the knowledge of the receiver and the parameters, and the method specification in implication form. Naively, one could expect that since the precondition is already in the obligation we could simply add the postcondition to the knowledge. This would be unsound. By using the specification in implication form, the system prevents circular reasoning: we could otherwise use the postcondition to prove the precondition. Instead, when the system shows that the precondition of S' holds, it can assume the postcondition of S'. Similar to logical *and* above, we are assuming that our parametric logic supports at least logical *implication*, but of course other forms of logical consequence could work too.

Note that the postcondition will contain information about the result of the method body as information on the `result` variable.

New: As usual, to type an object instantiation, we inductively type all the parameters. In this way we obtain all the types $C_1 \dots C_n$, all the knowledge $P_1 \dots P_n$, and all the verification obligations $P'_1 \dots P'_n$. As we did for METHOD we use fresh variables to be able to compose predicates.

As we mentioned above, we rely on abstract state operations to represent state: that is, all the abstract methods in C need to be of form S_i `method` $C_i\ x_i()$; where `this`.$x_i()$ returns the value of field x_i, that in turn was initialized with the result of expression e_i. The function *getters*(C) returns all methods of this form.

Knowledge P''_i contains the knowledge of P_i (from expression e_i) and it links such knowledge to the result of calling method `result`.$x_i()$, so that calling a getter on the created object will return the expected value. However, the information is conditional over verifying the precondition of such getter. Note that we do not need to add the knowledge of the postcondition of $x_i()$ here; this will be handled by the METHOD rule when $x_i()$ is called.

Knowledge P is simply merging the accumulated knowledge; while the final obligation in addition to merging the accumulated obligations also requires that the precondition of all the getters hold. In this way the getter preconditions behave like the precondition of the constructor. By requiring those preconditions, we ensure that we can call the getters on all the created objects.

Sub: The subsumption rule is standard. We allow subtyping between class names. Note that we do not apply weakening and strengthening of conditions here.

Besides of typing correct programs, the typing rules of Trait-CbC have the goal to verify the correctness of method implementations. The following rules check whether a method or a *Body* are correct. The check for a correct method declaration in MOK calls a program verifier to verify the correctness. We need just one verifier call for the verification of each method because the rules above collected all needed knowledge and obligations.

MOK: In MOK, we construct a Γ, and we type the method body, obtaining knowledge P and obligation P'. The program verifier will know the type information of Γ, the premise of the method, and the knowledge P, and will prove the obligation P' and the postcondition of the method. This verification in the typing rule is indicated by the keyword **verify**. Here, we use implication, but a different program verifier may use a different form of logical consequence. The program verifier can access the specification of all the other methods since we also provide the declaration table.

AbsMOK: Abstract methods are correctly typed.

$$\frac{Ds \vdash e \rightarrow e'}{Ds \vdash \mathcal{E}_v[e] \rightarrow \mathcal{E}_v[e']} \quad (Ctx)$$

$$\frac{S \;\texttt{method}\; C\; m(C_1\; x_1,\; \ldots,\; C_n\; x_n)\; e;\; \in\; methods(C)}{Ds \vdash \texttt{new}\; C(vs).m(v_1 \ldots v_n) \rightarrow e[\texttt{this} = \texttt{new}\; C(vs),\; x_1 = v_1,\; \ldots,\; x_n = v_n]} \quad (\text{MCALL})$$

$$\frac{abs(Ds(C)) = S_1 \;\texttt{method}\; C_1\; x_1();\; \ldots\; S_n \;\texttt{method}\; C_n\; x_n();}{Ds \vdash \texttt{new}\; C(v_1 \ldots v_n).x_i() \rightarrow v_i} \quad (\text{GETTER})$$

Figure 3: Reduction rules of TraitCbC

BodyTyped: A *Body* is correctly typed, if all the methods in the declaration of the *Body* are correctly typed.

4.2.3. *Reduction Rules.* We formulate three reduction rules for our system to evaluate input expressions to final values. We introduce an evaluation context $\mathcal{E}_v$ in our syntax in Fig. 1 to define the order of evaluation. The rules of Fig. 3 are explained in the following.

Ctx: This is the conventional contextual rule, allowing the execution of subexpressions.

Mcall: We reduce a method call to an expression e, where the receiver is replaced with `new` $C(vs)$, and each parameter x_i with the actual value v_i. We also ensure that the method is declared in the class C.

Getter: In our formalism, abstract methods without arguments represents getters. Notation *abs*(*Body*) returns the set of all abstract methods in *Body*. A valid class can only have abstract methods without arguments, and they will all represent getters.

4.2.4. *Flattening Semantics.* When we implement methods in several traits, we have to check that these traits are compatible when they are composed. This process to derive a complete class from a set of traits is called flattening. We follow the traditional flattening semantics [DNS+06]. A class that is defined by composing several traits is obtained by flattening rules. All methods are direct members of the class [DNS+06]. Overall, our flattening process works as a big step reduction arrow, where we reduce a trait expression into a well-typed and verified body.

To introduce our flattening rules in Fig 4, we first define the helper functions. The function *allMeth* collects all method headers with the same name as m in all input bodies (Definition 4.1). When two *Body*s are composed (Definition 4.2), the implemented interfaces are united and the methods are composed. The composition of methods (Definition 4.3) collects methods that are only defined in one of the input sets. If a method is in both sets, it is composed (Definition 4.4). Here, we distinguish four cases. If one method is abstract and the other is concrete, we have to show that the precondition of the abstract method implies the precondition of the concrete method. Additionally, the postcondition of the concrete one has to imply the postcondition of the abstract one. This is similar to Liskov's substitution principle [LW94]. The second case is the symmetric variant of the first case. In the third and fourth case, two abstract methods are composed. Here, the specification of one abstract method has to imply the specification of the other abstract method such that an implementation can still satisfy all specifications of abstract methods. If both methods are concrete, the composition is correctly left undefined. This composition error can be resolved by making one method m abstract in the *Body*, as defined in Definition 4.5. The resulting *Body* is similar with the difference that the implementation of the method m is omitted. The flattening rules in Fig. 4 are explained in the following in detail. In these rules, a set of traits is flattened to a declaration containing all methods. If abstract and concrete methods with the same name are composed, Definitions 4.2-4.4 are used to guarantee correctness of the composition.

Definition 4.1 (All Methods)**.** $\mathit{allMeth}(m,\ \mathit{Bodys}) =$
$\{\mathit{MH};\ \mid \mathit{Body}\ \in\ \mathit{Bodys},\ \mathit{Body}(m) = \mathit{MH};\}$

Definition 4.2 (Body Composition)**.** $\mathit{Body}_1 + \mathit{Body}_2 = \mathit{Body}$
$\{\texttt{interface}?\ [\mathit{Cs}_1]\ \mathit{Ms}_1\} + \{\texttt{interface}?\ [\mathit{Cs}_1]\ \mathit{Ms}_1\} =$
$\{\texttt{interface}?\ [\mathit{Cs}_1\ \cup \mathit{Cs}_2]\ \mathit{Ms}_1 + \mathit{Ms}_2\}$

Definition 4.3 (Methods Composition)**.** $\mathit{Ms}_1 + \mathit{Ms}_2 = \mathit{Ms}$
- $(M\ \mathit{Ms}_1) + \mathit{Ms}_2 = M\ (\mathit{Ms}_1 + \mathit{Ms}_2)$
  if $\mathit{methName}(M) \notin \mathit{dom}(\mathit{Ms}_2)$
- $(M_1\ \mathit{Ms}_1) + (M_2\ \mathit{Ms}_2) = M_1 + M_2\ (\mathit{Ms}_1 + \mathit{Ms}_2)$
  if $\mathit{methName}(M_1) = \mathit{methName}(M_2)$
- $\emptyset + \mathit{Ms} = \mathit{Ms}$

Definition 4.4 (Method Composition)**.** $M_1 + M_2 = M$
- S `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n)\ e;\ +\ S'$ `method` $C\ m(C_1\ \_ \ldots C_n\ \_);$
  $=\ S$ `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n)\ e;$
  if $\mathit{Pre}(S')$ *implies* $\mathit{Pre}(S)$ *and* $\mathit{Post}(S)$ *implies* $\mathit{Post}(S')$
- $\mathit{MH}_1;\ +\ \mathit{MH}_2\ e;\quad =\quad \mathit{MH}_2\ e;\ +\ \mathit{MH}_1;$
- S `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n);\ +\ S'$ `method` $C\ m(C_1\ \_ \ldots C_n\ \_);$
  $=\ S$ `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n);$
  if $\mathit{Pre}(S')$ *implies* $\mathit{Pre}(S)$ *and* $\mathit{Post}(S)$ *implies* $\mathit{Post}(S')$
- S `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n);\ +\ S'$ `method` $C\ m(C_1\ \_ \ldots C_n\ \_);$
  $=\ S'$ `method` $C\ m(C_1\ x_1 \ldots C_n\ x_n);$
  if $(\mathit{Pre}(S)$ *implies* $\mathit{Pre}(S')$ *and* $\mathit{Post}(S')$ *implies* $\mathit{Post}(S))$
  and not $(\mathit{Pre}(S')$ *implies* $\mathit{Pre}(S)$ *and* $\mathit{Post}(S)$ *implies*$\mathit{Post}(S'))$

Definition 4.5 (Body Abstraction)**.** $\mathit{Body}[\texttt{makeAbstract}\ m]$
$\{[\mathit{Cs}]\ \mathit{Ms}_1\ S$ `method` $C\ m(\mathit{Cxs})\_;\ \mathit{Ms}_2\}[\texttt{makeAbstract}\ m]$
$= \{[\mathit{Cs}]\ \mathit{Ms}_1\ S$ `method` $C\ m(\mathit{Cxs});\ \mathit{Ms}_2\}$

FlatTop: The first rule flattens a set of declarations $D_1 \ldots D_n$ to a set $D'_1 \ldots D'_n$. We express this rule in a non-computational way: we assume to know the resulting $D'_1 \ldots D'_n$, and we use them as a guide to compute them. Note that if there is a resulting $D'_1 \ldots D'_n$ then it is unique; flattening is a deterministic process and $D'_1 \ldots D'_n$ are used only to type check the results. They are not used to compute the shape of the flattened code.

Non computational rules like this are common with nominal type systems [IPW01] where the type signatures of all classes and methods can be extracted before the method bodies are verified.

DFlat: This rule flattens an individual definition by flattening the trait expression. When the flattening produces a class definition, we also check that the body denotes an instantiable class; a class whose only abstract methods are valid getters. The function $\mathit{abs}(\mathit{Body})$ returns the abstract methods.

$$\frac{D'_1 \ldots D'_n \vdash D_1 \Downarrow D'_1 \qquad \ldots \qquad D'_1 \ldots D'_n \vdash D_n \Downarrow D'_n}{D_1 \ldots D_n \Downarrow D'_1 \ldots D'_n} \quad (\textsc{FlatTop})$$

$$\frac{Ds;\ Name \vdash E \Downarrow Body \qquad \texttt{if}\ Name\ \texttt{of form}\ C\ \texttt{then}\ abs(Body) = S\ T\ x_1(); \ldots S\ T\ x_n();}{Ds \vdash Name = E \Downarrow Name = Body} \quad (\textsc{DFlat})$$

$$\frac{\begin{array}{c} Body = \{\texttt{interface}?\ [Cs]\ M_1 \ldots M_n\} \\ Body' = \{\texttt{interface}?\ [Cs]\ M_1 \ldots M_n\ Ms\} \\ Ms = \{\Sigma allMeth(Ds,\ Cs,\ m) \mid m \in dom(Cs)\ and\ m \notin dom(Body)\} \qquad Ds;\ Name \vdash Body' : OK \end{array}}{Ds;\ Name \vdash Body \Downarrow Body'} \quad (\textsc{BFlat})$$

$$\frac{}{Ds;\ Name \vdash t \Downarrow Ds(t)} \quad (\textsc{tFlat}) \qquad \frac{Ds;\ Name \vdash E_1 \Downarrow Body_1 \qquad Ds;\ Name \vdash E_2 \Downarrow Body_2}{Ds;\ Name \vdash E_1 + E_2 \Downarrow Body_1 + Body_2} \quad (+\textsc{Flat})$$

$$\frac{\begin{array}{c} Ds;\ Name \vdash E \Downarrow Body \qquad Body = \{[Cs]\ \overline{M}_1\ S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n)\_;\ \overline{M}_2\} \\ Body' = \{[Cs]\ \overline{M}_1\ S\ \texttt{method}\ C\ m(C_1\ x_1 \ldots C_n\ x_n);\ \overline{M}_2\} \end{array}}{Ds;\ Name \vdash E[\texttt{makeAbstract}\ m] \Downarrow Body'} \quad (\textsc{AbsFlat})$$

Figure 4: Flattening rules of TraitCbC

BFlat: It may look surprising that the *Body* does not flatten to itself. This represents what happens in most programming languages, where implementing an interface implicitly imports the abstract signature for all the methods of that interface. In the context of verification also the specification of such interface methods is imported. In concrete, *Body′* is like *Body*, but we add *Ms* by collecting all the methods of the interfaces that are not already present in the *Body*.

Moreover, we check that all the methods defined in the class respect the typing and the specification defined in the interfaces: if a class has S `method Foo foo();` or S `method Foo foo() e;` and there is a S' `method Foo foo();` in the interface, then S must respect the specification S'. The system then checks that the *Body* is well-typed and verified by calling $Ds;\ Name \vdash M_i : OK$

TFlat: A trait t is flattened to its declaration $Ds(t)$.

+Flat: The composition of two expression E_1 and E_2, where both expressions are first reduced to $Body_1$ and $Body_2$, results in the composition of these bodies as defined in Definition 4.2.

AbsFlat: An expression E where one method m is made abstract flattens to a *Body′*. We know that E flattens to *Body*. The only difference between *Body* and *Body′* is that the one method m is abstract in *Body′*. In *Body*, the method can be abstract or concrete.

4.2.5. *Soundness of* TraitCbC. In this subsection, we formulate the main result of the TraitCbC approach. We prove soundness of the flattening process with a parametric logic. We claim that if you have a language without code reuse and with sound and modular post-hoc verification then the language supports CbC simply by adding traits to the language. That is, traits intrinsically enable a CbC program construction approach.

To prove soundness of the construction approach of TraitCbC (Theorem 4.9: Sound CbC Process) as exemplified in Section 4.1, we have to show that the flattening process is correct (Theorem 4.8: General Soundness). In turn, to prove General Soundness, we need two lemmas which state that the composition of traits is correct (Lemma 4.6) and that a trait after the `makeAbstract` operation is still correct (Lemma 4.7).

In Lemma 4.6, we have well-typed definitions Ds, and two well-typed and verified traits in Ds, and the resulting trait/class is also well-typed and verified.

Lemma 4.6 (Composition correct)**.**
If $Ds(t1) = Body_1$*,* $Ds(t2) = Body_2$*,* $Ds(Name) = Body$*,* $Ds; t_1 \vdash Body_1 : OK$*,* $Ds; t_2 \vdash Body_2 : OK$*, and* $Body_1 + Body_2 = Body$*,*
then $Ds; Name \vdash Body : OK$

Proof. We prove by contradiction. We assume the resulting $Body$ is ill typed. By definition of BodyTyped, it means that one of the methods cannot be typed with either AbsMOK or MOK. The list of methods that need to be typed is obtained by Definition 4.2.

Abstract methods can only be typed with AbsMOK and are never wrong. Implemented methods can only be typed with MOK. If $\Gamma \vdash e : C \dashv P \models P'$ or the other precondition **verify** $Ds \vdash (\Gamma \ \&\ Pre(S) \ \&\ P) \models (P' \ \&\ Post(S))$ does not hold, it means that there was a method m_i with expression e_i in $Body_1$ (or symmetrically for $Body_2$) that was well-typed under $Ds; t_1 \vdash Body_1$. That means that all of its implemented methods were well-typed and verified. Typing e_i produces $P_i \models P'_i$ by using a Γ_{t1} containing $\texttt{this} : t_1$.

If $Ds; Name \vdash Body : OK$ is not applicable, the same expression e_i was typed using a Γ_{Name} containing $\texttt{this} : Name$. It produced $P''_i \models P'''_i$ so that **verify** $Ds \vdash (\Gamma_{Name} \ \&\ Pre(S) \ \&\ P''_i) \implies (P'''_i \ \&\ Post(S))$ does not hold. We know that **verify** $Ds \vdash (\Gamma_{t1} \ \&\ Pre(S) \ \&\ P_i) \implies (P'_i \ \&\ Post(S))$ holds by our assumption. By Definition 4.4, the contracts of the methods in $Body$ are simply stronger than the contracts of the methods in $Body_1$. The only difference between $P''_i \models P'''_i$ and $P_i \models P'_i$ is in the contracts of methods called on `this`. Assuming that our parametric logic implication is transitive, we know that **verify** $Ds \vdash (\Gamma_{t1} \ \&\ Pre(S) \ \&\ P_i) \implies (P'_i \ \&\ Post(S))$ entails **verify** $Ds \vdash (\Gamma_{Name} \ \&\ Pre(S) \ \&\ P''_i) \implies (P'''_i \ \&\ Post(S))$, thus we reach a contradiction. ☐

Lemma 4.7 shows that if we have a well-typed and verified trait, the operation `make−Abstract` results in a trait/class that is also well-typed and verified.

Lemma 4.7 (MakeAbstract correct)**.**
If $Ds(t) = Body$*,* $Ds(Name) = Body'$*,* $Ds; t \vdash Body : OK$*,*
and $Body[\texttt{makeAbstract}\ m] = Body'$*,*
then $Ds; Name \vdash Body' : OK$

Proof. We prove by contradiction. We assume the resulting $Body'$ is ill typed. By definition of BodyTyped, it means that one of the methods cannot be typed with either AbsMOK or MOK. The list of methods that need to be typed is obtained by Definition 4.2.

Abstract methods can only be typed with AbsMOK and are never wrong. We know that $Body$ is typable by our assumption. The only difference between $Body$ and $Body'$ is that the method m is made abstract. As we have seen for Lemma 4.6, we are typing $Body'$ in a different Γ. This case is even simpler than Lemma 4.6 because $Body$ and $Body'$ have exactly the same specifications. The abstract method m and thus $Body'$ cannot be ill typed. ☐

With these Lemmas, we can prove Theorem 4.8. Given a sound and modular verification language, then all programs that flatten are well-typed and verified. In a modular verification language, a method can be fully verified using only the information contained in the method declaration and the specification of any used method. Moreover, our parametric logic must support at least a commutative and associative *and* (but of course other ways to merge knowledge could work too) and a transitive *implication* (but of course other forms of logical consequence could work too).

Theorem 4.8 (General Soundness).
For all programs Ds where Ds flattens to Ds', and Ds' is well-typed; that is, fo rall $Name = Body \in Ds'$, we have $Ds';\ Name \vdash Body : OK$.

Proof. By induction on the size of Ds, and by induction on cases of E (the applied flattening rule for E).

- $Body$ only flattens if the $Body$ can be shown to be well-typed.
- t only reads a trait from the already verified Ds'.
- $Body_1 + Body_2$ is correct with Lemma 4.6. The lemma can be applied directly, if E is of depth one (e.g., $Body_1 + Body_2$). If E is more complex, we have to apply other cases of this case analysis.
- `makeAbstract` is handled similarly using Lemma 4.7.
- By the flattening relation, we know that $Body_1$ and $Body_2$ are well-typed in Ds. If we start from a program containing only well-typed and verified traits, any new class we can define by just composing those traits is well typed and verified. □

We now show that the TraitCbC approach is sound. Theorem 4.9 states that starting with one abstract method and a set of verified traits, the composed program is also verified.

Theorem 4.9 (Sound CbC Process).
Starting from a fully abstract specification t_0, and some construction steps $t_1 \ldots t_n$, we can write $C = t_0 + \cdots + t_n$ as our whole CbC approach, where $t_0 + t_1$ is the application of the first construction step. If we use CbC to construct programs, we can start from verified atomic units and get a verified result. Formally, if $t_0 = \{MH\}$ $t_1 = \{Ms_1\}$ $\ldots$ $t_n = \{Ms_n\}$ are well-typed, and

$$\begin{array}{lcl} t_0 = \{MH\} & & t_0 = \{MH\} \\ t_1 = \{Ms_1\}\ \ldots\ t_n = \{Ms_n\} & \Downarrow & t_1 = \{Ms_1\}\ \ldots\ t_n = \{Ms_n\} \\ C = t_0 + \cdots + t_n & & C = Body \end{array}$$

then $C = Body$ is well-typed.

Proof. This is a special case of Theorem 4.8. □

Theorem 4.9 shows clearly that trait composition intrinsically enables a CbC approach: An object-oriented programming language with traits and a corresponding specification language supports an incremental CbC approach.

4.3. Proof-of-Concept Implementation.

In this section, we describe the implementation, which instantiates TraitCbC in Java with JML [LBR98] as specification language and KeY [ABB+16] as verifier for Java code. Our trait implementation is based on interfaces with default implementation. Our open source tool is implemented in Java and integrated as plug-in in the Eclipse IDE.[6] Besides this prototype, other languages with a suitable verifier, such as Dafny [Lei10] and OpenJML [Cok11], can also be used to implement TraitCbC.

[6]Tool and evaluation at `https://doi.org/10.5281/zenodo.7766635`

In Listing 11, we show the concrete syntax of our implementation. Each method in a trait is specified with JML with the keywords `requires` and `ensures` for the pre- and postcondition. To verify the correctness of programs, we need two steps. First, we verify the correctness of a method implemented in a trait w.r.t. its specification. Second, for trait composition, our implementation checks the correct composition for all methods (cf. Definition 4.2). The syntax of trait composition is shown in Listing 12. In a tc-file (a file to specify the traits to be composed), the name of the resulting trait is given and the composed traits are connected with a plus operator. In Listing 12, trait `MaxElement1` is composed with trait `MaxElement2`. The trait `MaxElement2` must implement the methods `accessHead` and `maxTail`, so that we obtain a correct result in which all methods are implemented. To verify correctness of the trait composition, it is checked that the specification of a concrete method satisfies the specification of the abstract one with the same signature (cf. Definition 4.4). These verification goals are sent to KeY, which starts an automatic verification attempt.

```
public interface MaxElement1 {
/*@ requires list.size() > 0;
  @ ensures (\forall int n; list.contains(n);
  @ \result >= n) & list.contains(\result);
  @*/
  default public int maxElement(List list) {
    if (list.size() == 1) return accessHead(list);
    if (list.element() >= maxElement(list.tail()))
      { return accessHead(list); }
    else { return maxTail(list); } }

/*@ requires list.size() > 0;
  @ ensures \result == list.element();
  @*/
  public int accessHead(List list);

/*@ requires list.size() > 1;
  @ ensures (\forall int n; list.tail().contains(n);
  @ \result >= n) & list.tail().contains(\result);
  @*/
  public int maxTail(List list);
}
```

Listing 11: Example of a trait in our implementation

```
        ComposedMax = MaxElement1 + MaxElement2
```

Listing 12: Example of a trait composition

Evaluation. We evaluate our implementation by a feasibility study. First, we reimplemented an already verified case study in our trait-based language. We used the IntList [STAL11] case study, which is a small software product line (SPL) with a common code base and several features extending this code base. Here, we can show that our trait-based language also facilitates reuse. The IntList case study implements functionality to insert integers to a list in the base version. Extensions are the sorting of the list and different insert options (e.g., front /back). We implement five methods that exists in different variants with our trait-based CbC approach. We implement the case study in different granularities. The coarse-grained version is similar to the SPL implementation we started with [STAL11], confirming that traits are also amenable to implement SPLs as shown by Bettini et al. [BDS10]. The fine-grained version implements the five methods incrementally with 12 construction steps. We can reuse 6 of these steps during the construction of method variants.

We also implement three more case studies (BankAccount [TSAH12], Email [Hal05], and Elevator [PR01]) with TraitCbC and classic CbC to show that it is feasible to implement object-oriented programs with both approaches. We used CorC [RSC+19] as an instance of a classic CbC tool. We were able to implement 9 classes and verify 34 methods with a size of 1–20 lines of code. For future work, a user study is necessary to evaluate the usability of TraitCbC in comparison to classic CbC to empirically confirm our stated advantages.

5. The Different CbC-based Program Construction Approaches in Comparison

In this section, we discuss classic CbC in comparison to CbC-Block and TraitCbC. In Table 1, we summarize how the three approaches compare regarding main aspects of developing correct programs using tool support. The aspects comprise the programming language, the tool support, the procedure to develop programs, and the verification of the program.

Language. All approaches need an underlying programming and specification language. The defined refinement rules of the classic CbC approach are external to a programming language. That means, each refinement rule introduces some statement of the programming language by transforming the program. With CbC-Block and the block-instantiation rule more than one statement of the language can be introduced at once. TraitCbC is usable with languages that have traits. Methods can be implemented as defined by the language. No refinement rules are necessary.

Tool Support. Tool support is helpful for any of the approaches. For classic CbC, mostly pen and paper is used. There are a few specialized tools such as CorC [RSC+19], ArcAngel [OCW03], and SOCOS [Bac09, BEM07]. These tools force a certain programming procedure on the user because refinement rules must be applied to implement programs. CbC-Block is implemented in CorC and extends the set of refinement rules with the new rules for blocks. To verify the correctness of block instantiations, program verifiers can be reused. There are program verifiers for many languages, such as Java [ABB+16], C [CDH+09], and C# [BFL+11, BLS04]. Other languages are integrated with their verifier from the start, e.g., Spec# [BLS04] and Dafny [Lei10]. For TraitCbC, we also need a program verifier to prove the correctness of method implementations, but we do not need specialized tools to construct methods, such as CorC.

Construction Rules. To construct a program, classic CbC has a strict concept of refinement rules that must be applied to construct a program. CbC-Block relaxes this strict guideline to construct programs. Programs can be constructed stepwise as with classic CbC, but if desired, any number of refinement steps can be condensed with the block rules. In the extreme case, a whole program can be developed in one step. TraitCbC offers this

	Classic CbC	CbC-Block	TraitCbC
Language	Additional refinement rules for a programming language. Needs specification language.	Additional refinement rules for a programming language. Introduces a specified block of statements. Needs specification language.	Programming language with traits. Needs specification language.
Tool support	Pen and paper. Some specialized tools available.	Pen and paper. Some specialized tools available. Block instantiation rule relies on post-hoc verification tools.	Relies on post-hoc verification tools.
Construction Rules	Specific refinement rules.	Specific refinement rules.	Construction by composition of traits.
Correctness/ Debugging	Guarantees the correctness of each refinement step.	Guarantees the correctness of each refinement step. Refinements can be condensed with the block rules.	Guarantees the correctness of each construction step. Each method is specified so that each constructed method can directly be verified.
Proof complexity	Many, but small proofs.	Any granularity of proofs.	Any granularity of proofs.
Reuse	Refinement steps cannot be reused; only fully implemented methods can.	Refinement steps cannot be reused; only fully implemented methods can.	Each verified method in a trait can be reused.
Applications	Focuses on small, but correctness-critical algorithms.	Focuses on correctness-critical algorithms.	As TraitCbC is based on post-hoc verification, it can be used in similar areas where post-hoc verification is used. Traits are beneficial for incremental development and software product lines.

Table 1: Comparison of TraitCbC with CbC-Block and classic CbC

flexibility to construct programs without the need of external refinement rules. Methods can be developed freely and only need to be composed with respect to their specification. Nevertheless, TraitCbC supports to construct code in fine-grained steps, which are more amenable for verification than more complex methods.

Correctness/Debugging. Classical CbC gives explicit information about the program states before and after execution of each statement by the Hoare triple notation. The correctness of each applied refinement step is guaranteed by proving the side conditions of the refinement rule. Some side conditions are not directly provable because abstract statements in Hoare triples must be concretized first. In the worst case, a problem in the program is found only after some refinement steps. The abstract statements in classic CbC are not explicitly specified by the developer. Additional specifications in classic CbC are introduced with some rules such as an intermediate condition in the composition rule. Then, these specifications are propagated through the program to be constructed. Again, due to a possible delayed check of a side condition, a wrong specification is found only after some refinement steps.

If errors occur in the program development process, TraitCbC gives early and detailed information on the level of verified methods. By specifying the method under development and any abstract method that is called by this method, we can directly verify the correctness of the method under development. We assume that the introduced abstract methods will be correctly implemented in further refinement steps. With each step, the developer gets closer to the solution until finally all abstract methods are implemented. CbC-Block combines the characteristics of the other two approaches. The refinement rules of classic CbC can be applied, or blocks of statements can be introduced. The specified block is verified similar to a method in TraitCbC.

Proof Complexity. Classical CbC requires many small proofs to guarantee the correctness of a program. CbC-Block can condense the proofs into larger proofs using the block refinement rules. TraitCbC can have the same granularity and also the same proof effort as classic CbC, since each method implementation can correspond to just one refinement step. The advantage of TraitCbC and CbC-Block is that developers can freely implement a method body or a block. They must not stick to the same granularity as in the classic CbC refinement rules. Proof complexity can be balanced against verifier calls.

Reuse. A fully refined method can be reused in all approaches. For TraitCbC, we can easily reuse even very small units of code, since they are represented as methods in the traits. In classic CbC and CbC-Block, no refinement step can be reused.

Applications. The classic CbC approach does not scale well to development procedures for complete software system. Rather, individual algorithms can be developed with CbC [WKSC16]. With the block rules, the scalability is improved because refinement steps that are easy to prove can be combined into one block. This saves the application of refinement rules and their corresponding correctness proofs. With ArchiCorC [KRS20], we can even scale CbC to the development of correct component-based architectures. By composing components specified with required and provided interfaces, we support the creation of software architectures correct by construction.

As soon as we scale TraitCbC to real languages, we have the same application scenarios as approaches that already use post-hoc program verification. As argued by Damiani et al. [DDJS14], traits enable an incremental process of specifying and verifying software. Bettini et al. [BDS10] proposed to use traits for software product line development and highlighted the benefits of fine-grained reuse mechanisms. Here, TraitCbC's guideline is suitable for constructing new product lines step by step from the beginning.

Since CbC-Block extends classic CbC and can be freely applied at any granularity of refinement steps, we propose to use CbC-Block for any implementation of correctness-critical software, but the CbC approach must be well understood by the developer to be efficiently usable. In TraitCbC, methods are developed and composed directly, so less knowledge is needed to apply the approach, but developers can fall back into a post-hoc verification process and thus lose the benefits of CbC (e.g., if the developers first develop all methods and do not directly prove the correctness). In general, both approaches are usable for program development and the right choice depends on the preferences and prior knowledge of the developers.

Summary. In summary, TraitCbC and CbC-Block allow more flexible program construction without losing the advantages of incremental correct-by-construction program development. CbC-Block loosens the strict guideline of classic CbC by adding the block refinement rules. CbC-Block still needs specialized tools, such as CorC to be applicable. TraitCbC enables a CbC approach for trait-based languages without introducing refinement rules. This program construction approach combined with the flexibility of traits allows correct methods to be developed in small and reusable steps. TraitCbC is independent of special CbC tools and requires only a program verifier.

6. Related Work

In the following, we discuss related work for specifying and verifying software. We discuss related correctness-by-construction approaches and compare CorC with other tools for CbC.

Contracts and Program Verification. The implementation of CbC in CorC and the implementation of TraitCbC use JML, Java DL and Java to specify and write programs. For the verification, KeY [ABB$^+$16] is integrated in the backend. KeY is a deductive program verifier for Java programs specified with JML. In an intermediate step, the specified programs are translated to Java DL. Similarly, OpenJML [Cok11] verifies Java programs specified with JML. Besides Java/JML, many languages support pre-/postcondition contracts or other forms of specification to state program behavior. First, the programming language Eiffel introduced contracts and supported the design-by-contract approach [Mey88, Mey92]. Eiffel is an object-oriented programming language, where classes are specified with invariants, and methods with pre-/postconditions contracts. For the verification, AutoProof [KRMJ16, TFNP15] is integrated that translates the specified program to a logic formula. Then, an SMT-solver proves the validity of the formulas. For C#, the language Spec# is an extension to introduce contracts and invariants [BLS04, BFL$^+$11]. The verification is done by translating the proof obligations to an intermediate language BoogiePL that can be verified with Boogie [BCD$^+$05]. For the C language, the VCC [CDH$^+$09] and Frama-C [CKK$^+$12] tools verify annotated C code. VCC reuses the Spec# tool chain. For Java and C, the VeriFast [JSP10] tool verifies C and Java programs. VerCors [ABD$^+$14] also support the verification of C and Java programs with a focus on concurrent and distributed software. Another language with integrated specifications and verification is Dafny [Lei10]. Dafny is a functional language, but supports the compilation to other languages such as C#, Java, Go, and Python. Similarly, Whiley [PG13] is a designed language with associated verifier to simplify the verification of programs. The languages SPARK [Bar03] supports a subset of the Ada language to specify and verify Ada programs. In contrast to JML, the specification is not written as comments, but the Ada *aspect*-syntax is used to express contracts. The focus of all these languages and verification tools is the specification of program behavior and the verification that a program satisfies its specification. With CbC (CbC-Block and TraitCbC), we put the correct construction of programs in the foreground, instead of just verifying the correctness post-hoc. However, Watson et al. [WKSC16] argue that correctness-by-construction and post-hoc verification can be used together to combine their mutual strengths.

To verify trait languages, Damiani et al. [DDJS14] added specifications of methods in traits to verify correct trait composition. They proposed a modular and incremental verification process. Traits are introduced in many languages to support clean design and reuse, for example Smalltalk [DNS$^+$06], Java [BMN14] by utilizing default methods in interfaces, and other Java-like languages [BDSS13, LS08, SD05]. None of these trait languages were used to formulate a CbC approach to create correct programs. They only focus on code reuse or post-hoc verification.

Refinement-Based Correctness-by-Construction. The main idea of correctness-by-construction is the stepwise construction of a program from a starting specification with correctness guarantees for each step. We focused on correctness-by-construction by Kourie and Watson [KW12] that we called classic CbC. This classic CbC approach is based on Dijkstra [Dij76] and Gries [Gri87]. In this paragraph, we discuss related refinement-based

CbC approaches. All of these approaches create correct programs by refining an abstract program or system to a concrete implementation. This is the main difference to the composition-based CbC approach of TraitCbC, where atomic units of code are composed to whole programs.

Morgan's refinement calculus [Mor94] is similar to correctness-by-construction by Kourie and Watson [KW12]. Both approaches have the same theoretical foundation, but Morgan's refinement calculus is more elaborated with a large number of different refinement rules, where many rules are only formally interesting. Kourie and Watson [KW12] reduced the refinement rules to a minimal but sufficient set, such that CbC becomes comprehensible for developers without a major background in formal methods. The language ArcAngel [OCW03] with the verifier ProofPower [ZOC09] implements Morgan's refinement calculus. The tool uses a tactic language to apply a sequence of refinement rules for program refinement. Thus, a tactic has the same benefit as the application of a block refinement in CbC-Block because the application of refinement rules is condensed to one refinement step. The difference is that for an introduced block of code in CbC-Block, it does not matter what classic CbC refinement rules would have to be applied to introduce that block of code. A tactic still applies the refinement rules stated in that tactic sequentially.

The invariant based programming [BW12, Bac09] shifts the focus from pre-/postcondition contracts as starting point for refinements to invariants. The tool SOCOS [Bac09, BEM07] implements Back's methodology. Similar to CorC, SOCOS has a graphical user interface to create a program in the form of a UML-style state chart. Refinement steps introduce new states and transitions in the state chart and check compliance with the invariants. A completely refined program is proved correct and executable code can be generated. In CorC, the graphical user interface present the refinement steps in a hierarchical tree structure that more directly represent the structure of the code (comparable with an abstract syntax tree). Therefore, CorC and also the implementation of TraitCbC are on the level of source code.

Further refinement-based methodologies are Event-B [Abr10, ABH$^+$10] for automata-based systems and Circus [OCW09, OGC08] for state-rich reactive systems. Both methodologies work on an abstraction level with abstract models instead of specified source code. In refinement steps these abstract models of the system are transformed to concrete and executable implementation. Here, each refined result guarantees conformations with the initial model. Event-B is supported by the tool Rodin [ABH$^+$10], and Circus is supported by the tool CRefine [OGC08]. The main difference to CbC by Kourie and Watson [KW12], and TraitCbC is the abstraction level. We specify and verify source code rather than automata-based systems.

Data refinement [HKKN13, HL22, LT12, CDM13] is a related approach that focuses on the refinement of (abstract) programs with abstract types to correct and more efficient programs with concrete types. Haftmann et al. [HKKN13] examine how the Isabelle/HOL code generator applies data refinements to produce executable versions of abstract programs. Cohen et al. [CDM13] present an approach to refine Coq programs to enhance computational efficiency. Haslbeck and Lammich [HL22] not only ensure functional correctness during data refinement, they also verify worst-case complexity of algorithms at the LLVM level. The main difference to CbC by Kourie and Watson [KW12] is that data refinement approaches start with algorithms on abstract data structures that are refined to more concrete data structures, whether CbC by Kourie and Watson focuses on the incremental development of the algorithm itself. Therefore, both approaches can used in concert to develop more efficient algorithm.

Extensions to Correctness-by-Construction and CorC. CorC has been extended in several directions to allow the structured program development for larger software systems and further application areas. With ArchiCorC [KRS20], we integrate the construction of correct software architectures. We bundle CorC programs into reusable software components. The components communicate via required and provided interfaces where ArchiCorC guarantees the compatibility between them. With VarCorC [BRS20] software product lines are developed correct by construction. A software product lines is used to systematically construct a family of similar software programs instead of developing monolithic programs. VarCorC ensures the correctness of all possible software variants of the product line. In addition to functional correctness, correctness-by-construction and CorC are extended to guarantee nonfunctional properties. As a first example, we introduced CbC refinement rules to ensure that programs [RKTS20, RKS+22] follow an information flow policy which defines the allowed flow of information in a program. In every refinement step, security and functional correctness of the program is guaranteed, such that insecure and incorrect programs are prohibited by construction. The goal of these extensions is that program development in CorC is scalable and that CbC can be used for additional application areas. Orthogonally, this article focuses on improving the flexibility of developing programs correct by construction (e.g., by introducing the block refinement rules).

Program and Specification Synthesis. Program synthesis is a technique that generates programs from user given specifications automatically. Pioneers in this field are Manna et al. [MW80]. Gulwani et al. [GPS+17] give an overview of state-of-the-art program synthesis approaches. For example, for Fortran, Stickel et al. [SWL+94] deductively extract programs from user-given graphical specifications. They compose procedures from libraries to full implementations. Similarly, Gulwani et al. [GJTV10] synthesize programs by composing base components from a specified library. Polikarpova et al. [PKSL16] synthesize recursive programs from specifications by utilizing type information. Similarly, synthesis of function summaries [Hoa71, CDK+15, SFS12] automatically generate pre-/postcondition specifications from programs to achieve modular verification and to improve verification time. With CbC (classic CbC, CbC-Block, or TraitCbC), developers have the task to specify and create programs according to that specification. Therefore, CbC is a program development approach where the developer determines the resulting program, while program synthesis generates one of possibly many programs that fulfills the specification. Contrary to this, the synthesis of a function summary generates one of possibly many specifications for a program. Synthesis has scalability limitations due to an enormous search space of programs/specifications and ambiguity of user intent.

7. Conclusion

In this article, we presented CbC-Block and TraitCbC two incremental program construction approaches that guide developers to implementations that are correct by construction. CbC-Block extends classic CbC with block refinement rules. These rules allow to condense the application of CbC refinement rules into one block refinement. Thus, CbC-Block increase flexibility in the development of programs because any sequence of statements can be introduced in a block, while still ensuring the correctness of that introduced block.

TraitCbC uses method calls and trait composition instead of refinement rules to guarantee functional correctness. We formalize the concept of a trait-based object-oriented language with a parametric specification language to allow a broader range of languages to adopt this concept. The main advantage of TraitCbC is the simplicity of the refinement process that supports code reuse. We compared classic CbC, CbC-Block, and TraitCbC qualitatively with regard to their programming constructs, tool support, and usability. CbC-Block and TraitCbC both relax the strict guideline of CbC without losing the benefits of a constructive program construction approach.

As future work, user studies could be conducted with all three approaches to further evaluate the usability of the approaches. We want to investigate how the more flexible construction approaches of TraitCbC and CbC-Block are received by developers. We also want to compare the development times and potential types of programming errors between the approaches. These user studies will help to develop concrete guidelines on which approach is appropriate under which circumstances and with which team.

Acknowledgments. This work was partially supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03). We thank Frederik Fröling for his work on CbC-Block in his Master's Thesis.

References

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. *Deductive Software Verification–The KeY Book: From Theory to Practice*, volume 10001. Springer, 2016.

[ABD+14] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer, 2014.

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[Bac09] Ralph-Johan Back. Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.

[Bar84] Hendrik P Barendregt. *The Lambda Calculus*, volume 3. North-Holland Amsterdam, 1984.

[Bar03] John Gilbert Presslie Barnes. *High Integrity Software: The Spark Approach to Safety and Security.* Pearson Education, 2003.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCK+22] Tabea Bordis, Loek Cleophas, Alexander Kittelmann, Tobias Runge, Ina Schaefer, and Bruce W. Watson. Re-CorC-ing KeY: Correct-by-Construction Software Development Based on KeY. In *The Logic of Software. A Tasting Menu of Formal Methods.* Springer, 2022.

[BDS10] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing Software Product Lines Using Traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2096–2102, 2010.

[BDSS13] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. TRAITRECORDJ: A Programming Language with Traits and Records. *Science of Computer Programming*, 78(5):521–541, 2013.

[BEM07] Ralph-Johan Back, Johannes Eriksson, and Magnus Myreen. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *International Conference on Tests and Proofs (TAP)*, volume 4454 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2007.

[BFL$^+$11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communication of the ACM*, 54(6):81–91, June 2011.

[BLS04] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[BMN14] Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-Oriented Programming in Java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 181–186, 2014.

[BRS20] Tabea Bordis, Tobias Runge, and Ina Schaefer. Correctness-by-Construction for Feature-Oriented Software Product Lines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 22–34. ACM, 2020.

[BW12] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction.* Springer Science & Business Media, 2012.

[CDH$^+$09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[CDK$^+$15] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Synthesising Interprocedural Bit-Precise Termination Proofs. In *International Conference on Automated Software Engineering (ASE)*, pages 53–64. IEEE, 2015.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs: Third International Conference*, pages 147–162. Springer, 2013.

[Cha06] Roderick Chapman. Correctness by Construction: A Manifesto for High Integrity Software. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 43–46, 2006.

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[Cok11] David R Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods Symposium*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer, 2011.

[DDJS14] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. Verifying Traits: An Incremental Proof System for Fine-Grained Reuse. *Formal Aspects of Computing*, 26(4):761–793, 2014.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communication of the ACM*, 18(8):453–457, August 1975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DNS$^+$06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[GJTV10] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component Based Synthesis Applied to Bitvector Programs. Technical report, Citeseer, 2010.

[GPS$^+$17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[Gri87] David Gries. *The Science of Programming.* Springer, 1987.

[Hal05] Robert J Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[HC02] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1):18–25, 2002.

[HKKN13] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data Refinement in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 100–115. Springer, 2013.

[HL22] Maximilian PL Haslbeck and Peter Lammich. For a few dollars more: Verified fine-grained algorithm analysis down to llvm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–36, 2022.

[Hoa71] Charles Antony Richard Hoare. Procedures and Parameters: An Axiomatic Approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer, 1971.

[IPW01] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages And Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.

[KFFD86] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[KRMJ16] Mansur Khazeev, Victor Rivera, Manuel Mazzara, and Leonard Johard. Initial Steps Towards Assessing the Usability of a Verification Tool. In *International Conference in Software Engineering for Defence Applications*, volume 717 of *Advances in Intelligent Systems and Computing*, pages 31–40. Springer, 2016.

[KRS20] Alexander Knüppel, Tobias Runge, and Ina Schaefer. Scaling Correctness-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*, pages 187–207. Springer, 2020.

[KW12] Derrick G Kourie and Bruce W Watson. *The Correctness-by-Construction Approach to Programming.* Springer Science & Business Media, 2012.

[LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420. Citeseer, 1998.

[Lei95] K Rustan M Leino. *Toward Reliable Modular Programs.* California Institute of Technology, 1995.

[Lei10] K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[LS08] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–32, 2008.

[LT12] Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving: Third International Conference, ITP 2012*, pages 166–182. Springer, 2012.

[LW94] Barbara H Liskov and Jeannette M Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[Mey88] Bertrand Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 2nd edition, 1994.

[MW80] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, jan 1980.

[OCW03] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and Jim Woodcock. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28–47, 2003.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP Semantics for Circus. *Formal Aspects of Computing*, 21(1):3–32, 2009.

[OGC08] Marcel Vinicius Medeiros Oliveira, Alessandro Cavalcante Gurgel, and C G Castro. CRefine: Support for the Circus Refinement Calculus. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290. IEEE, Nov 2008.

[PG13] David J Pearce and Lindsay Groves. Whiley: A Platform for Research in Software Verification. In *International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 238–248. Springer, 2013.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. *ACM SIGPLAN Notices*, 51(6):522–538, aug 2016.

[PR01] Malte Plath and Mark Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[RBTS21] Tobias Runge, Tabea Bordis, Thomas Thüm, and Ina Schaefer. Teaching Correctness-by-Construction and Post-hoc Verification–The Online Experience. In *Formal Methods Teaching Workshop*, volume 13122 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2021.

[RKS$^+$22] Tobias Runge, Alexander Kittelmann, Marco Servetto, Alex Potanin, and Ina Schaefer. Information Flow Control-by-Construction for an Object-Oriented Language. In *International Conference on Software Engineering and Formal Methods*, volume 13550 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2022.

[RKTS20] Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. Lattice-Based Information Flow Control-by-Construction for Security-by-Design. In *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering*, pages 44–54. ACM, 2020.

[RPTS22] Tobias Runge, Alex Potanin, Thomas Thüm, and Ina Schaefer. Traits: Correctness-by-Construction for Free. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, volume 13273 of *Lecture Notes in Computer Science*, pages 131–150. Springer, 2022.

[RSC$^+$19] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W Watson. Tool Support for Correctness-by-Construction. In *International Conference on Fundamental Approaches to Software Engineering*, volume 11424 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2019.

[RTC$^+$19] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W Watson. Comparing Correctness-by-Construction with Post-Hoc Verification - A Qualitative User Study. In *Formal Methods. FM 2019 International Workshops. Refine*, volume 12233 of *Lecture Notes in Computer Science*, pages 388–405. Springer, 2019.

[SD05] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-Like Languages. In *European Conference on Object-Oriented Programming*, pages 453–478, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[SFS12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-Based Function Summaries in Bounded Model Checking. In *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2012.

[SH19] Dominic Steinhöfel and Reiner Hähnle. Abstract Execution. In *International Symposium on Formal Methods*, pages 319–336. Springer, 2019.

[STAL11] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic Detection of Feature Interactions Using the Java Modeling Language: An Experience Report. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[SWL$^+$94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In *International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 1994.

[tBCSW18] Maurice H. ter Beek, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. X-by-Construction. In *International Symposium on Leveraging Applications of Formal Methods*, pages 359–364, Cham, 2018. Springer International Publishing.

[TFNP15] Julian Tschannen, Carlo A Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.

[TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-Based Deductive Verification of Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, page 11–20, New York, NY, USA, 2012. Association for Computing Machinery.

[WKSC16] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience? In *International Symposium on Leveraging Applications of Formal Methods*, volume 9952 of *Lecture Notes in Computer Science*, pages 730–748. Springer, 2016.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[ZOC09] Frank Zeyda, Marcel Oliveira, and Ana Cavalcanti. Supporting ArcAngel in ProofPower. *Electronic Notes in Theoretical Computer Science*, 259:225–243, 2009.

Appendix A. Interview Questions

(1) Which task was more difficult and why?
(2) Which tasks were solved?
(3) What were the biggest problems during the development?
(4) Is the development according to CbC understandable?
(5) Is the use of the block rules understandable?
(6) Is the introduction of the block rules reasonable?
(7) Would you use the block rules when implementing according to CbC?
(8) How do you like the development in the textual editor?
(9) How do you like the development in the graphical editor?
(10) Is the textual or the graphical editor preferred?
(11) Which elements from the editors are particularly helpful or inadequate and why?
(12) What functionalities are still missing in the editors?
(13) What would it take for you to develop according to CbC in your workday?