Tool Support for Correctness-by-Construction

Tobias Runge[1], Ina Schaefer[1], Loek Cleophas[2,3], Thomas Thüm[1], Derrick Kourie[3,4], and Bruce W. Watson[3,4]

[1] Software Engineering, TU Braunschweig, Germany
[2] Software Engineering Technology, TU Eindhoven, The Netherlands
[3] Information Science, Stellenbosch University, South Africa
[4] Centre for Artificial Intelligence Research, CSIR, South Africa
{tobias.runge,i.schaefer,t.thuem}@tu-bs.de,
{loek,bruce,derrick}@fastar.org

Abstract. Correctness-by-Construction (CbC) is an approach to incrementally create formally correct programs guided by pre- and postcondition specifications. A program is created using refinement rules that guarantee the resulting implementation is correct with respect to the specification. Although CbC is supposed to lead to code with a low defect rate, it is not prevalent, especially because appropriate tool support is missing. To promote CbC, we provide tool support for CbC-based program development. We present CorC, a graphical and textual IDE to create programs in a simple while-language following the CbC approach. Starting with a specification, our open source tool supports CbC developers in refining a program by a sequence of refinement steps and in verifying the correctness of these refinement steps using the theorem prover KeY. We evaluated the tool with a set of standard examples on CbC where we reveal errors in the provided specification. The evaluation shows that our tool reduces the verification time in comparison to post-hoc verification.

1 Introduction

Correctness-by-Construction (CbC) [12, 13, 19, 23] is a methodology to construct formally correct programs guided by a specification. CbC can improve program development because every part of the program is designed to meet the corresponding specification. With the CbC approach, source code is incrementally constructed with a low defect rate [19] mainly based on three reasons. First, introducing defects is hard because of the structured reasoning discipline that is enforced by the refinement rules. Second, if defects occur, they can be tracked through the refinement structure of specifications. Third, the trust in the program is increased because the program is developed following a formal process [14].

Despite these benefits, CbC is still not prevalent and not applied for large-scale program development. We argue that one reason for this is missing tool support for a CbC-style development process. Another issue is that the programmer mindset is often tailored to the prevalent post-hoc verification approach.

CbC has been shown to be beneficial even in domains where post-hoc verification is required [29]. In post-hoc verification, a method is verified against pre- and postconditions. In the CbC approach, we refine the method stepwise, and we can check the method partially after each step since every statement is surrounded by a pair of pre- and postconditions. The verification of refinement steps and Hoare triples reduces the proof complexity since the proof task is split into smaller problems. The specifications and code developed using the CbC approach can be used to bootstrap the post-hoc verification process and allow for an easier post-hoc verification as the method constructed using CbC generally is of a structure that is more amenable to verification [29].

In this paper, we present CorC,[5] a tool designed to develop programs following the CbC approach. We deliberately built our tool on the well-known post-hoc verifier KeY [4] to profit from the KeY ecosystem and future extensions of the verifier. We also add CbC as another application area to KeY, which opens the possibility for KeY users to adopt the CbC approach. This could spread the constructive CbC approach to areas where post-hoc verification is prevalent.

Our tool CorC offers a hybrid textual-graphical editor to develop programs using CbC. The textual editor resembles a normal programming editor, but is enriched with support for pre- and postcondition specifications. The graphical editor visualizes the code, its specification, and the program refinements in a tree-like structure. The developers can switch back and forth between both views. In order to support the correct application of the refinement rules, the tool is integrated with KeY [4] such that proof obligations can be immediately discharged during program development. In a preliminary evaluation, we found benefits of CorC compared to paper-and-pencil-based application of CbC and compared to post-hoc verification.

2 Foundations of Correctness-by-Construction

Classically, CbC [19] starts with the specification of a program as a Hoare triple comprising a precondition, an abstract statement, and a postcondition. Such a triple, say T, should be read as a total correctness assertion: if T is in a state where the precondition holds and its abstract statement is executed, then the execution will terminate and the postcondition will hold. T will be true for a certain set of concrete program instantiations of the abstract program and false for other instantiations. A refinement of T is a triple, say T', which is true for a subset of concrete programs that render T to be true.

In our work, pre-/post-condition specifications for programs are written in *first-order logic* (FOL). A formula in FOL consists of atomic formulas which are logically connected. An atomic formula is a predicate which evaluates to true or false. Programs in this work are written in the CorC language, which is inspired by the *Guarded Command Language* (GCL) [11] and presented below.

[5] https://github.com/TUBS-ISF/CorC, CorC is an acronym for Correctness-by-Construction

$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$	*can be refined to*
1. Skip :	$\{\mathtt{P}\}$ *skip* $\{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}$
2. Assignment :	$\{\mathtt{P}\}\ x := E\ \{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{Q}[\mathtt{x} := \mathtt{E}]$
3. Composition :	$\{\mathtt{P}\}\ \mathtt{S1}\,;\,\mathtt{S2}\ \{\mathtt{Q}\}$ *iff there is an intermediate condition* $\mathtt{M}$ *such that* $\{\mathtt{P}\}\ \mathtt{S1}\ \{\mathtt{M}\}$ *and* $\{\mathtt{M}\}\ \mathtt{S2}\ \{\mathtt{Q}\}$
4. Selection :	$\{\mathtt{P}\}$ **if** $\mathtt{G}_1 \to \mathtt{S}_1$ **elseif** $\ldots\ \mathtt{G}_n \to \mathtt{S}_n$ **fi** $\{\mathtt{Q}\}$ *iff* ($\mathtt{P}$ *implies* $\mathtt{G}_1 \vee \mathtt{G}_2 \vee \ldots \vee \mathtt{G}_n$) *and* $\{\mathtt{P} \wedge \mathtt{G}_i\}\ \mathtt{S}_i\ \{\mathtt{Q}\}$ *holds for all* i.
5. Repetition :	$\{\mathtt{P}\}$ **do** $[\mathtt{I}, \mathtt{V}]\ \mathtt{G} \to \mathtt{S}$ **od** $\{\mathtt{Q}\}$ *iff* ($\mathtt{P}$ *implies* $\mathtt{I}$) *and* ($\mathtt{I} \wedge \neg\mathtt{G}$ *implies* $\mathtt{Q}$) *and* $\{\mathtt{I} \wedge \mathtt{G}\}\ \mathtt{S}\ \{\mathtt{I}\}$ *and* $\{\mathtt{I} \wedge \mathtt{G} \wedge \mathtt{V}{=}\mathtt{V}_0\}\ \mathtt{S}\ \{\mathtt{I} \wedge \mathtt{0}{\leq}\mathtt{V} \wedge \mathtt{V}{<}\mathtt{V}_0\}$
6. Weaken pre :	$\{\mathtt{P}'\}\ S\ \{\mathtt{Q}\}$ *iff* $\mathtt{P}$ *implies* $\mathtt{P}'$
7. Strengthen post :	$\{\mathtt{P}\}\ S\ \{\mathtt{Q}'\}$ *iff* $\mathtt{Q}'$ *implies* $\mathtt{Q}$
8. Subroutine :	$\{\mathtt{P}\}$ *Sub* $\{\mathtt{Q}\}$ *with subroutine* $\{\mathtt{P}'\}$ *Sub* $\{\mathtt{Q}'\}$ *iff* $\mathtt{P}$ *is equal to* $\mathtt{P}'$ *and* $\mathtt{Q}'$ *is equal to* $\mathtt{Q}$

Fig. 1. Refinement Rules in CbC [19]

For the concrete instantiation of conditions and assignments, our tool uses a host language. We decided for Java, but other languages are also possible.

To create programs using CbC, we use refinement rules. A Hoare triple is refined by applying rules, which introduce CorC language statements, so that a concrete program is created. The concrete program obtained by refinement is guaranteed to be correct by construction, provided that the correctness-preserving refinement steps have been accurately applied. In Figure 1, we present the statements and refinement rules used in CbC and our tool.

Skip. A skip or empty statement is a statement that does not alter the state of the program (i.e., it does nothing) [11, 19]. This means a Hoare triple with a skip statement evaluates to true if the precondition implies the postcondition.

Assignment. An assignment statement assigns an expression of type T to a variable, also of type T. In the tool, we use a Java-like assignment ($\mathtt{x} = \mathtt{y}$). To refine a Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ with an assignment statement, the assignment rule is used. This rule replaces the abstract statement $\mathtt{S}$ by an assignment $\{\mathtt{P}\}\ \mathtt{x} = \mathtt{E}\ \{\mathtt{Q}\}$ iff $\mathtt{P}$ implies $\mathtt{Q}[\mathtt{x} := \mathtt{E}]$.

Composition. A composition statement is a statement which splits one abstract statement into two. A Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ is split to $\{\mathtt{P}\}\ \mathtt{S}_1\ \{\mathtt{M}\}$ and $\{\mathtt{M}\}\ \mathtt{S}_2\ \{\mathtt{Q}\}$ in which $\mathtt{S}$ is refined to $\mathtt{S1}$ and $\mathtt{S2}$. $\mathtt{M}$ is an intermediate condition which evaluates to true after $\mathtt{S1}$ and before $\mathtt{S2}$ is executed [11].

Selection. Selection in our CorC language works as a switch statement. It refines a Hoare triple $\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ to $\{\mathtt{P}\}$ **if** $\mathtt{G}_1 \to \mathtt{S}_1$ **elseif** $\ldots\ \mathtt{G}_n \to \mathtt{S}_n$ **fi** $\{\mathtt{Q}\}$. The guards $\mathtt{G}_i$ are evaluated, and the sub-statement $\mathtt{S}_i$ of the *first* satisfied guard is executed. We use a switch-like statement so that every sub-statement has an associated guard for further reasoning. The selection refinement rule can only be

used if the precondition P implies the disjunction of all guards so that at least one sub-statement could be executed.

Repetition. The repetition statement $\{\mathtt{P}\}$ **do** $[\mathtt{I}, \mathtt{V}]$ $\mathtt{G} \rightarrow$ $\mathtt{S}$ **od** $\{\mathtt{Q}\}$ works like a while loop in other languages. If the loop guard G evaluates to true, the associated loop statement S is executed. The repetition statement is specified with an invariant I and a variant V. To refine a Hoare triple $\{\mathtt{P}\}$ S $\{\mathtt{Q}\}$ with a repetition statement, (1) the precondition P has to imply the invariant I of the repetition statement, (2) the conjunction of invariant and the negation of the loop guard G have to imply the postcondition Q, and (3) the loop body has to preserve the invariant by showing that $\{\mathtt{I} \wedge \mathtt{G}\}$ S $\{\mathtt{I}\}$ holds. To verify termination, we have to show that the variant V monotonically decreases in each loop iteration and has 0 as a lower bound.

Weaken precondition. The precondition of a Hoare triple can be weakened if necessary. The weaken precondition rule replaces the precondition P with a new one $\mathtt{P}'$ only if P implies $\mathtt{P}'$ [12].

Strengthen postcondition. To strengthen a postcondition, the strengthen postcondition rule can be used. A postcondition Q is replaced by a new one $\mathtt{Q}'$ only if $\mathtt{Q}'$ implies Q [12].

Subroutine. A subroutine can be used to split a program into smaller parts. We use a simple subroutine call where we prohibit side effects and parameters. A triple $\{\mathtt{P}\}$ S $\{\mathtt{Q}\}$ can be refined to a subroutine $\{\mathtt{P}'\}$ *Sub* $\{\mathtt{Q}'\}$, if the precondition $\mathtt{P}'$ of the subroutine is equal to the precondition P of the refined statement and the postcondition $\mathtt{Q}'$ of the subroutine is equal to the postcondition Q of the refined statement. The subroutine can be constructed as a separate CbC program to verify that it satisfies the specification. The Hoare triple $\{\mathtt{P}'\}$ *Sub* $\{\mathtt{Q}'\}$ is the starting point to construct a program using CbC.

3 Correctness-by-Construction by Example

To introduce the programming style of CbC, we demonstrate the construction of a linear search algorithm using CbC [19]. The linear search problem is defined as follows: We have an integer array a of some length, and an integer variable x. We try to find an element in the array a which has the same value as the variable x, and we return the index i where the (last) element x was found, or -1 if the element is not in the array.

To construct the algorithm, we start with concretizing the pre- and postcondition of the algorithm. Before the algorithm is executed, we know that we have an integer array. Therefore, we specify $\mathtt{a} \neq \mathtt{null} \wedge \mathtt{a.length} \geq 0$ as precondition P. The postcondition forces that if the index i is greater than or equal to zero, the element is found on the returned index i ($\mathtt{Q} := (\mathtt{i} \geq 0 \implies \mathtt{a}[\mathtt{i}] = \mathtt{x})$).

Our algorithm traverses the array in reverse order and checks for each index whether the value is equal to x. In this case, the index is returned. To create this algorithm, we construct an invariant I for the loop:

$$\mathtt{I} := \neg\mathtt{appears}(\mathtt{a}, \mathtt{x}, \mathtt{i}+1, \mathtt{a.length}) \wedge \mathtt{i} \geq -1 \wedge \mathtt{i} < \mathtt{a.length}$$

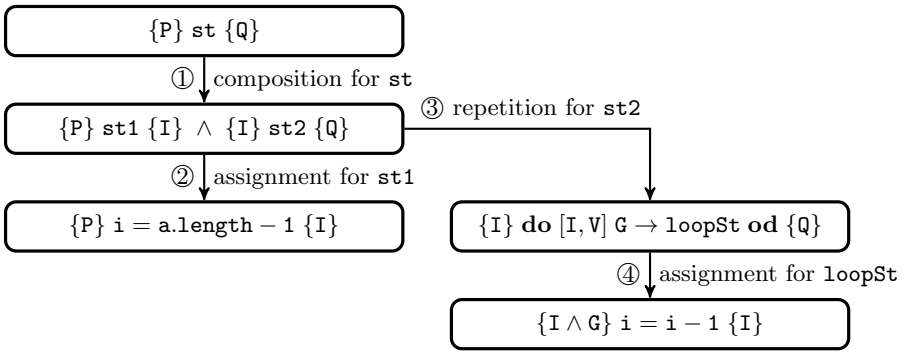

Fig. 2. Refinement steps for the linear search algorithm

The invariant is used to split the array into two parts. A part from $\texttt{i}+1$ to `a.length` where `x` is not contained, and a part from zero to `i` which is not checked yet. In every iteration, the next index of the array is checked. The predicate $\texttt{appears}(\texttt{a},\texttt{x},\texttt{l},\texttt{h})$ asserts that `x` occurs in array `a` inside the range from `l` (included) to `h` (excluded). The predicate can be translated to FOL as $\exists \texttt{i} : (\texttt{i}{\geq}\texttt{l} \wedge \texttt{i}{<}\texttt{h} \wedge \texttt{a}[\texttt{i}]{=}\texttt{x})$.

We can use the CbC refinement rules to implement linear search. The refinement steps for the example are shown in Figure 2 and numbered from ① to ④. To create a loop in the program, we need to initialize a loop counter variable to establish the invariant. Therefore, we split the program by introducing a composition statement (① in Figure 2). The invariant `I` is used as intermediate condition (i.e., $\texttt{M} := \texttt{I}$), because it has to be true after the initialization, and before the first loop step. The statement `st1` is refined to an assignment statement ②. We initialize `i` with $\texttt{a.length}-1$ to start at the end of the array. This assignment satisfies the intermediate condition `I` where `i` is replaced by $\texttt{a.length}-1$. The range of `appears` is empty, and therefore the predicate evaluates to true. To refine the second statement (`st2`), we use the repetition refinement rule ③. As long as `x` is not found, we iterate through the array. As guard of the repetition, we use $(\texttt{i}{\geq}0 \wedge \texttt{a}[\texttt{i}]{\neq}\texttt{x})$. The invariant of the repetition is the invariant `I` introduced above. The variant `V` is $\texttt{i}+1$. To verify that this refinement is valid, we have to verify that the precondition of the repetition statement implies the invariant, and that the invariant and the negated guard imply the postcondition of the repetition (cf. Rule 5). Both are valid because the precondition is equal to the invariant and the postcondition of the repetition statement (in this case it is `Q`) is equal to the negated guard. The last step is to refine the abstract loop statement (`loopSt`) ④. We use an assignment to decrease `i` and get the final program. We can verify that the invariant holds after each loop iteration. The program terminates because the variant decreases in every step and it is always greater than or equal to zero.

4 Tool Support in CorC

CorC extends KeY's application area by enabling CbC to spread the constructive engineering to areas where post-hoc verification is prevalent. KeY programmers can use both approaches to construct formally correct programs. By using CorC, they develop specification and code that can bootstrap the post-hoc verification. The CorC tool[6] is realized as an Eclipse plug-in in Java. We use the Eclipse Modeling Framework (EMF)[7] to specify a CbC meta model. This meta model is used by two editor views, a textual and a graphical editor. The Hoare triple verification is implemented by the deductive program verification tool KeY [4]. In the following list, we summarize the features of CorC.

- Programs are written as Hoare triple specifications, including pre-/postcondition specifications and abstract statements or assignment/skip statements in concrete triples.
- CorC has eight rules to construct programs: skip, assignment, composition, selection, repetition, weakening precondition, strengthening postcondition, and subroutine (cf. Section 2).
- Pre-/postconditions and invariant specification are automatically propagated through the program.
- CorC comprises a graphical and a textual editor that can be used interchangeably.
- Up to now, CorC supports integers, chars, strings, arrays, and subroutine calls without side effects, I/O, and library calls.
- Hoare triples are typically verified by KeY automatically. If the proof cannot be closed automatically, the user can interact with KeY.
- Helper methods written in Java 1.5 can be used in a specification.
- CorC comprises content assist and an automatic generation of intermediate conditions.

4.1 Graphical Editor

The graphical editor represents CbC-based program refinement by a tree structure. A node represents the Hoare triple of a specific CorC language statement. Figure 3 presents the linear search algorithm of Section 3 in the graphical editor. The structure of the tree is the same as in Figure 2. The additional nodes on the right specify used program variables including their type and global invariant conditions. The global invariant conditions are added to every pre- and postcondition of Hoare triples to simplify the construction of the program. In the example, we specify the array `a` and the range of variable `i` to support the verification, as KeY requires this range to be explicit for verification.

The root node of the tree shows the abstract Hoare triple for the overall program with a symbolic name for the abstract statement. In every node, the

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

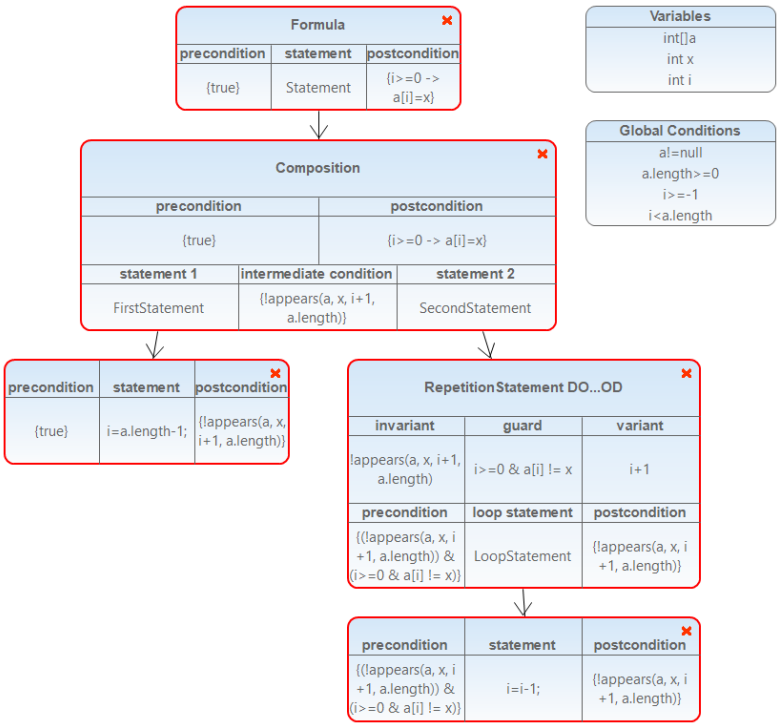

Fig. 3. Linear search example in the graphical editor

pre- and postcondition are specified on the left and right of the node under the corresponding header. A composition statement node, the second statement of the tree, contains the pre- and postcondition and additionally defines an intermediate condition. The intermediate condition is the middle term in the bottom line. Both abstract sub-statements of the composition have a symbolic name and can be further refined by adding a connection to another node (i.e., creating a parent-child relation). The repetition node contains fields to specify the invariant, the guard and the variant of the repetition. These fields are in the middle row. The pre- and postcondition are associated to the inner loop statement. An assignment node (cf. both leaf nodes of the figure) contains the precondition, the assignment, and the postcondition. The representations of the nodes for the refinements not illustrated in this example are similar.

Refinement steps are represented by edges. The pre- and postconditions are propagated from parents to their children on drawing the parent/child relation. We explicitly show the propagated conditions in a node to improve readability. The propagated conditions from the parent are unmodifiable because refinement rules determine explicitly how conditions are propagated. An exception are the rules to weaken the precondition or strengthen the postcondition. Here, the conditions can be overridden. At the repetition statement, we only depict the

pre-/postconditions of the inner loop statement to reduce the size of this node. The pre-/postconditions of the parent node (in our example the composition statement) are not shown explicitly, but they are propagated internally to verify that the repetition refinement rule is satisfied. To visualize the verification status, the nodes have a green border if proven, a red one otherwise.

By showing the Hoare triples explicitly, problems in the program can be localized. If some leaf node cannot be proven, the user has to check the assignment and the corresponding pre-/postcondition. If an error occurred, the conditions on the refinement path up to pre-/postcondition of the starting Hoare triple can be altered. Other paths do not need to be checked. To prove the program correct, we have to prove that the refinement is correct. Aside from the side conditions of refinement rules (cf. iff conditions in refinement rules), only the leaf nodes of the refinement tree which contain basic Hoare triples with skip or assignment statements need to be verified by a prover, while all composite statements are correct by construction of their conditions.

To support the user in developing intermediate conditions for composition statements, our tool can compute the weakest precondition from a postcondition and a concrete assignment by using the KeY theorem prover. So, the user can create a specific assignment statement and generate the intermediate conditions afterwards. We also support modularization, to cover cases where algorithms become too large. Sub-algorithms can be created using CbC in other CorC programs. We introduce a simple subroutine rule which can be used as a leaf node in the editor. The subroutine has a name and it is connected to a second diagram with the same name as the subroutine. This subroutine call is similar to a classic method call. It can be used to decompose larger CbC developments to multiple smaller programs.

4.2 Textual Editor

The textual editor is an editor for the CorC programming language described above. The user writes code by using keywords for the specific statements and enriches the code with conditions, such as invariants or intermediate conditions, and assignments in our CorC syntax. The syntax of the composed statements in the textual editor is shown in Figure 4. In the `GlobalConditions` declaration, we enumerate the needed global conditions separated with a comma. The used variables are enumerated after the `JavaVariables` keyword.

The linear search example program presented in Section 3 is shown in the syntax of CorC in Listing 1. The program starts with keyword `Formula`. The pre- and postcondition of the abstract Hoare triple are written after the `pre:` and `post:` keywords. The abstract statement of the Hoare triple is refined to a composition statement in lines 3–16. The statements are surrounded by curly brackets to establish the refinement structure. We have the first statement in lines 4–6, the intermediate condition in line 7 and the second statement in lines 8–15. The first statement is refined to an assignment (Line 5). The refinement is done by introducing an assignment in Java syntax (`i = a.length − 1;`). The second statement is refined to a repetition statement (cf. the syntax of a repetition statement

Selection statement	*Repetition statement*
if (“guard”) **then** {statement}	**while** (“guard”)
elseif (“guard”) **then** {statement}	**inv**: [“invariant”] **var**: [“variant”]
...	**do** {statement} **od**
fi	

Fig. 4. Syntax of statements in textual editor

```
1  Formula "linearSearch"
2  pre: {"true"}
3  {
4    {
5      i=a.length-1;
6    }
7    intm: ["!appears(a, x, i+1, a.length)"]
8    {
9      while ("i>=0 & a[i]!=x")
10     inv: ["!appears(a, x, i+1, a.length)"]
11     var: ["i+1"] do
12     {
13       i=i-1;
14     } od
15   }
16 }
17 post: {"i>=0 -> a[i]=x"}
18
19 GlobalConditions
20   conditions {"a!=null", "a.length>=0",
21     "i>=-1", "i<a.length"}
22
23 JavaVariables
24   variables {"int[] a", "int x", "int i"}
```

Listing 1. Linear search example in the textual editor

in Figure 4). We specify the guard, the invariant, and the variant. Finally, the single statement of the loop body is refined to an assignment in Line 13.

As in the graphical editor, pre-/postconditions are propagated top-down from a parent to a child statement. For example, the intermediate condition of a composition statement which is the postcondition of the first sub-statement and the precondition of the second, appears only once in the editor (e.g., Line 7). To support the user, we implemented syntax highlighting and a content assist. When starting to write a statement, a user may employ auto-completion where the statements are inserted following the syntax in Figure 4. The user can specify the

```
1 \javaSource "src";
2 \include "helper.key";
3 \programVariables {int x;}
4 \problem {
5   (x = 0) -> \<{x=x+1;}\> (x = 1)
6 }
```

Listing 2. KeY problem file

conditions, then the next statement can be refined. The editor also automatically checks the syntax and highlights syntax errors. Information markers are used to indicate statements which are not proven yet. For example, the Hoare triple of the assignment statement ($\texttt{i} = \texttt{a.length} - \texttt{1}$) in Listing 1 has to be verified, and CorC marks the statement according to the proof completion results.

4.3 Verification of CorC Programs

To prove the refined program is correct, we have to prove side conditions of refinements correct (e.g., prove that an assignment satiesfies the pre-/postcondition specification). This reduces the proof complexity because the challenge to prove a complete program is decomposed into smaller verification tasks. The intermediate Hoare triples are verified indirectly through the soundness of the refinement rules and the propagation of the specifications from parent nodes to child nodes [19]. Side conditions occur in all refinements (cf. iff conditions in refinement rules). These side conditions, such as the termination of repetition statements or that at least one guard in a selection has to evaluate to true, are proven in separate KeY files.

For the proof of concrete Hoare triples, we use the deductive program verifier KeY [4]. Hoare triples are transformed to KeY's dynamic logic syntax. The syntax of KeY problem files is shown in Listing 2. Using the keyword `javaSource`, we specify the path to Java helper methods which are called in the specifications. These methods have to be verified independently with KeY. A KeY helper file, where the users can define their own FOL predicates for the specification, is included with the keyword `include`. For example, in CorC a predicate $appears(a, x, l, h)$ (cf. the linear search example) can be used which is specified in the helper file as a FOL formula. The variables used in the program are listed after the keyword `programVariables`. After `problem`, we define the Hoare triple to be proven, which is translated to dynamic logic as used by KeY. KeY problem files are verified by KeY. As we are only verifying simple Hoare triples with skip or assignment statements, KeY is usually able to close the proofs automatically if the Hoare triple is valid.

To verify total correctness of the program, we have to prove that all repetition statements terminate. The termination of repetition statements is shown by proving that the variants in the program monotonically decrease and are bounded. Without loss of generality, we assume this bound to equal 0, as this

is what KeY requires. This is done by specifying the problem in the KeY file in the following way: `(invariant & guard) -> {var0:=var} \<{std}\> (invariant & var<var0 & var>=0)`. The code of the loop body is specified at `std` to verify that after one iteration of the loop body the variant `var` is smaller than before but greater than or equal to zero.

To verify Hoare triples in the graphical editor, we implemented a menu entry. The user can right-click on a statement and start the automatic proof. If the proof is not closed, the user can interact with the opened KeY interface. To prove Hoare triples in the textual editor, we automatically generate all needed problem files for KeY whenever the user saves the editor file. The proof of the files is started using a menu button. The user gets feedback which triples are not proven by means of markers in the editor.

4.4 Implementation as Eclipse Plugin

We extended the Eclipse modeling framework with plugins to implement the two editors. We have created a meta model of the CbC language to represent the required constructs (i.e., statements with specification). The statements can be nested to create the CbC refinement hierarchy. The graphical and the textual editor are projections on the same meta model. The graphical editor is implemented using the framework Graphiti.[8] It provides functionality to create nodes and to associate them to domain elements, such as statements and specifications. The nodes can be added from a palette at the side of the editor, so no incorrect statement with its associated specification can be created. We implemented editing functionality to change the text in the node; the background model is changed simultaneously. Graphiti also provides the possibility to update nodes (e.g., to propagate pre- and postconditions), if we connect those nodes by refinement edges. The refinement is checked for compliance with the CbC rules.

The textual editor is implemented using XText.[9] We created a grammar covering every statement and the associated specification. If the user writes a program, the text is parsed and translated to an instance of the meta model. If a program is created in one editor, a model (an instance of our meta model) of the program is created in the background. We can easily transform one view into the other. The transformation is a generation step and not a live synchronization between both views, but it is carried out invisibly for the user when changing the views.

In implementing CorC, we considered the exchangeability of the host language. The specifications and assignments are saved as strings in the meta model. They are checked by a parser to comply with Java. This parser could be exchanged to support a different language. The verification is done by generating KeY files which are then evaluated by KeY. Here, we have to exchange the generation of the files if another theorem prover should be integrated. The information of the meta model may have to be adopted to fit the needs of the other prover. We also have to implement a programmatic call to the other prover.

[8] https://eclipse.org/graphiti/
[9] https://eclipse.org/Xtext/

Algorithm	#Nodes in GE	#Lines in TE	#Lines with JML	#Verified CorC triples	CbC Total Proof-Nodes	CbC Total Proof-Time	PhV Total Proof-Nodes	PhV Total Proof-Time
Linear Search	5	12	10	5/5	285	0.4 s	589	1.2 s
Max. Element	9	21	15	9/9	1023	1.2 s	993	1.8 s
Pattern Matching	14	23	20	13/13	21131	54.9 s	201619	1479.3 s
Exponentiation	7	21	17	7/7	6588	15.2 s	7303	20.4 s
Log. Approx.	5	16	12	5/5	13756	42.7 s	18835	68.5 s
Dutch Flag	8	26	24	8/8	4107	5.7 s	4993	13.4 s
Factorial	5	15	13	4/4	1554	3.6 s	1598	4.4 s

(GE) Grahical Editor, (TE) Textual Editor, (PhV) Post-hoc Verification

Table 1. Evaluation of the example programs

5 Evaluation

The tool support offers new chances to evaluate CbC versus post-hoc verification. We quantitatively compare the development and verification of programs with CorC and with post-hoc verification. This is to check the hypothesis that the verification of algorithms is faster with CorC than with post-hoc verification. We created the first eight algorithms from the book by Kourie and Watson [19] in our graphical editor. For comparison purposes, we also wrote each example as a plain Java program with JML specifications in order to directly verify it with KeY. The specifications are the same as in CorC. We measured the verification time and the proof nodes that KeY needed to close the proofs for both approaches. The results of the evaluation are presented in Table 1(verification time rounded).

The algorithms have 5 to 14 nodes in the graphical editor and 12 to 26 lines of code in the textual editor. The Java version with a JML specification always has fewer lines (between 8% and 29% smaller). The additional specifications, such as the intermediate conditions of composition statements, and the global invariant conditions and variables cause more lines of code in the CbC program.

The verification of the eight algorithms worked nearly without problems. We verified 7 out of 8 examples within CorC. In the cases without problems, every Hoare triple and the termination of the loops could be proven. We had to prove fewer Hoare triples than nodes in the editor, as not every node has

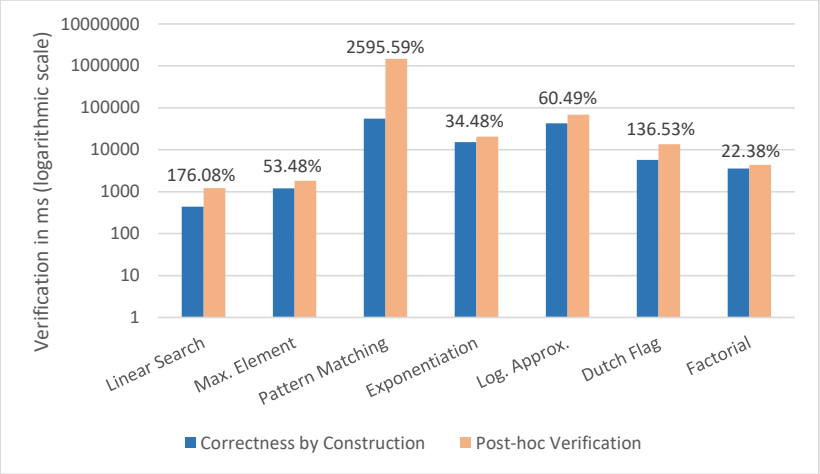

Fig. 5. Proof time of CbC and post-hoc verification in logarithmic scale

to be proven separately. Composition nodes are proven indirectly through the refinement structure. For *exponentiation*, *logarithm*, and *factorial*, we had to implement recursive helper methods which are used in the specification. Therefore, the programs impose upper bounds for integers to shorten the proof. The *binary search* algorithm could not be verified automatically in KeY using post-hoc verification or CorC. In each step, when the element is not found, the algorithm halves the array. KeY could not prove that the searched element is in the new boundaries because verification problems with arithmetic division are hard to prove for KeY automatically.

In the case of measured proof nodes, *maximum element* needs slightly fewer nodes proved with post-hoc verification than with CbC. In the other cases, the proofs for the algorithms constructed with CbC are 3% to 854% smaller. The largest difference was measured for the *pattern matching* algorithm. The proof is reduced to a ninth of the nodes.

The verification time is visualized in Figure 5. The time is measured in milliseconds and scaled logarithmically. The proofs for the CbC approach are always faster showing lower proof complexity. For *maximum element*, *exponentiation*, *logarithm* and *factorial*, the post-hoc verification time requires between 22% and 60% more time. The difference increases for *Dutch flag* and *linear search* to 137% and 176%, respectively. Algorithm *pattern matching* has the biggest difference. Here, the CbC approach needs nearly a minute, but the post-hoc approach needs over 24 minutes. To verify our hypothesis, we apply the non-parametric paired Wilcoxon-Test [30] with a significance level of 5%. We can reject the null hypothesis that CbC verification and post-hoc verification have no significant difference in verification time (p-value = 0.007813). This rejection of the null hypothesis in an empirical evidence for our hypothesis that verification is faster with CorC than with post-hoc verification.

With our tool support, we were able to compare the CbC approach with post-hoc verification. For our examples, we evaluated that the verification effort is reduced significantly which indicates a reduced proof complexity. It is worthwhile to further investigate the CbC approach, also to profit from synergistic effects in combination with post-hoc verification. As we built CorC on top of KeY, the post-hoc verification of programs constructed with CorC is feasible.

An advantage of CorC is the overview on all Hoare triples during development. In this way, we found some specifications where descriptions in the book by Kourie and Watson [19] were not precise enough to verify the problem in KeY. For example, in the *pattern matching* algorithm, we had to verify two nested loops. At one point, we had to verify that the invariant of the inner loop implies the invariant of the outer loop. This was not possible, so we extended the invariant of the inner loop to be the conjunction of both invariants. In the book of Kourie and Watson [19], this conjunction of both invariants was not explicitly used.

6 Related Work

We compare CorC to other programming languages and tools using specification or refinements. The programming language Eiffel is an object-oriented programming language with a focus on design-by-contract [21, 22]. Classes and methods are annotated with pre-/postconditions and invariants. Programs written in Eiffel can be verified using AutoProof [18, 28]. The verification tool translates the program with assertions to a logic formula. An SMT-solver proves the correctness and returns the result. Spec# is a similar tool for specifying C# programs with pre-/postcondition contracts. These programs can be verified using Boogie. The code and specification is translated to an intermediate language (BoogiePL) and verified [5, 6]. VCC [8] is a tool to annotate and verify C code. For this purpose, it reuses the Spec# tool chain. VeriFast [16] is another tool to verify C and Java programs with the help of contracts. The contracts are written in separation logic (a variant of Hoare logic). As in Eiffel, the focus of Spec#, VCC, and VeriFast is on post-hoc verification and debugging failed proof attempts.

The Event-B framework [2] is a related CbC approach. Automata-based systems including a specification are refined to a concrete implementation. Atelier B [1] implements the B method by providing an automatic and interactive prover. Rodin [3] is another tool implementing the Event-B method. The main difference to CorC is that CorC works on code and specifications rather than on automata-based systems.

ArcAngel [25] is a tool supporting Morgan's refinement calculus. Rules are applied to an initial specification to produce a correct implementation. The tool implements a tactic language for refinements to apply a sequence of rules. In comparison to our tool, ArcAngel does not offer a graphical editor to visualize the refinement steps. Another difference is that ArcAngel creates a list of proof obligations which have to be proven separately. CRefine [26] is a related tool for the Circus refinement calculus, a calculus for state-rich reactive systems. Like

our tool, CRefine provides a GUI for the refinement process. The difference is that we specify and implement source code, but they use a state-based language. ArcAngelC [10] is an extension to CRefine which adds refinement tactics.

The tools iContract [20] and OpenJML [9] apply design-by-contract. They use a special comment tag to insert conditions into Java code. These conditions are translated to assertions and checked at runtime which is a difference to our tool because no formal verification is done. DBC-Python is a similar approach for the Python language which also checks assertions at runtime [27].

To verify the CbC program, we need a theorem prover for Hoare triples, such as KeY [4]. There are other theorem provers which could be used (e.g., Coq [7] or Isabelle/HOL [24]). The Tecton Proof System [17] is a related tool to structure and interactively prove Hoare logic specification. The proofs are represented graphically as a set of linked trees. These interactive provers do not fit our needs because we want to automate the verification process. KeY provides a symbolic execution debugger (SED) that represents all execution paths with specifications of the code to the verification [15]. This visualization is similar to our tree representation of the graphical editor. The SED can be used to debug a program if an error occur during the post-hoc verification process.

7 Conclusion and Future Work

We implemented CorC to support the Correctness-by-Construction process of program development. We created a textual and a graphical editor that can be used interchangeably to enable different styles of CbC-based program development. The program and its specification are written in one of the editors and can be verified using KeY. This reduces the proof complexity with respect to post-hoc verification. We extended the KeY ecosystem with CorC. CorC opens the possibility to utilize CbC in areas where post-hoc verification is used as programmers could benefit from synergistic effects of both approaches. With tool support, CbC can be studied in experiments to determine the value of using CbC in industry.

For future work, we want to extend the tool support, and we want to evaluate empirically the benefits and drawbacks of CorC. To extend the expressiveness, we implement a rule for methods to use method calls in CorC. These methods have to be verified independently by CorC/KeY. We could investigate whether the method call rules of KeY can be used for our CbC approach. Another future work is the inference of conditions to reduce the manual effort. Postconditions can be generated automatically for known statements by using the strongest postcondition calculus. Invariants could be generated by incorporating external tools. As mentioned earlier, other host languages and other theorem provers can be integrated in our IDE.

The second work package for future work comprise the evaluation with a user study. We could compare the effort of creating and verifying algorithms with post-hoc verification and with our tool support. The feedback can be used to improve the usability of the tool.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International journal on software tools for technology transfer 12(6), 447–466 (2010)
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification–The KeY Book: From Theory to Practice, vol. 10001. Springer (2016)
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and Verification: The Spec# Experience. Communication of the ACM 54(6), 81–91 (Jun 2011)
6. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. pp. 49–69. Springer (2004)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer Science & Business Media (2013)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
9. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. In: NASA Formal Methods Symposium. Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer (2011)
10. Conserva Filho, M., Oliveira, M.V.M.: Implementing Tactics of Refinement in CRefine. In: International Conference on Software Engineering and Formal Methods. Lecture Notes in Computer Science, vol. 7504, pp. 342–351. Springer (2012)
11. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communication of the ACM 18(8), 453–457 (Aug 1975)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
13. Gries, D.: The Science of Programming. Springer (1987)
14. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software 19(1), 18–25 (2002)
15. Hentschel, M.: Integrating Symbolic Execution, Debugging and Verification. Ph.D. thesis, Technische Universität Darmstadt (2016)
16. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Asian Symposium on Programming Languages And Systems. Lecture Notes in Computer Science, vol. 6461, pp. 304–311. Springer (2010)
17. Kapur, D., Nie, X., Musser, D.R.: An Overview of the Tecton Proof System. Theoretical Computer Science 133(2), 307–339 (1994)
18. Khazeev, M., Rivera, V., Mazzara, M., Johard, L.: Initial Steps Towards Assessing the Usability of a Verification Tool. In: International Conference in Software Engineering for Defence Applications. Advances in Intelligent Systems and Computing, vol. 717, pp. 31–40. Springer (2016)
19. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer Science & Business Media (2012)

20. Kramer, R.: iContract-The Java Design by Contract Tool. In: Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. NO 98EX176). pp. 295–307. IEEE (Aug 1998)
21. Meyer, B.: Eiffel: A Language and Environment for Software Engineering. Journal of Systems and Software 8(3), 199–246 (1988)
22. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (1992)
23. Morgan, C.: Programming from Specifications. Prentice Hall, 2nd edn. (1994)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer Science & Business Media (2002)
25. Oliveira, M.V.M., Cavalcanti, A., Woodcock, J.: ArcAngel: A Tactic Language for Refinement. Formal Aspects of Computing 15(1), 28–47 (2003)
26. Oliveira, M.V.M., Gurgel, A.C., Castro, C.G.: CRefine: Support for the Circus Refinement Calculus. In: 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods. pp. 281–290. IEEE (Nov 2008)
27. Plosch, R.: Tool Support for Design by Contract. In: Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176). pp. 282–294. IEEE (Aug 1998)
28. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9035, pp. 566–580. Springer (2015)
29. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience? In: International Symposium on Leveraging Applications of Formal Methods. Lecture Notes in Computer Science, vol. 9952, pp. 730–748. Springer (2016)
30. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Science & Business Media (2012)