Correctness-by-Construction and Post-hoc Verification: A Marriage of Convenience?

Bruce W. Watson[1,2](✉), Derrick G. Kourie[1,2], Ina Schaefer[3], and Loek Cleophas[1,4]

[1] Department of Information Science, Stellenbosch University, Stellenbosch, South Africa
{bruce,derrick,loek}@fastar.org

[2] Centre for Artificial Intelligence Research, CSIR Meraka Institute, Pretoria, South Africa

[3] Technische Universität Braunschweig, Software Engineering Institute, Braunschweig, Germany
i.schaefer@tu-bs.de

[4] Technische Universiteit Eindhoven, Software Engineering and Technology Group, Eindhoven, The Netherlands

Abstract. Correctness-by-construction (CbC), traditionally based on weakest precondition semantics, and post-hoc verification (PhV) aspire to ensure functional correctness. We argue for a lightweight approach to CbC where lack of formal rigour increases productivity. In order to mitigate the risk of accidentally introducing errors during program construction, we propose to complement lightweight CbC with PhV. We introduce lightweight CbC by example and discuss strength and weaknesses of CbC and PhV and their combination, both conceptually and using a case study.

1 Introduction

In today's world, software that controls safety-, mission- and business-critical applications is pervasive. Test-first programming [1], requirements or code coverage-based testing, adherence to coding standards and reliance on software patterns are examples of common practices aimed at satisfying functional requirements. To avoid injury, loss of life or unmanageable follow-up costs resulting from such systems, much greater confidence in the functional correctness of the software is required than is demanded of more mundane software applications [2]. Hence, to complement common software engineering practices, more rigorous development approaches are needed. These may include adherence to standards such as DO178-B for avionics or ISO26262 for automotive applications. Formal methods such as formal program verification [3] may also be used.

The starting point for formal program verification, also called post-hoc verification (PhV), is an already written program. Annotations that capture the functional requirements are added to the program. These are typically in the form of a pre-/postcondition specification of each method in a class. Additionally, invariants for the class may be provided. In order to be able to prove automatically that

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 730–748, 2016.
DOI: 10.1007/978-3-319-47166-2_52

the code adheres to these specifications, auxiliary loop annotations have to be provided, expressing loop invariants and variants. A PhV tool (such as KeY [4], VeriFast [5], Spec# [6] or Krakatoa/Why [7]) then uses a formal calculus to establish correctness of the program with respect to its pre-/postconditions and invariants. Such a tool also uses the variants of the program's loops to verify that the program terminates. However, PhV is not yet widely practiced. One reason is the limited set of program constructs supported by program verification tools (e.g. dynamic arrays and pointers are notoriously difficult to support). Another reason is that it may be very challenging to provide the annotations needed, especially if the program to be verified is poorly structured.

In pursuit of functional correctness, we propose the adoption of a *lightweight* version of the approach to software construction that was pioneered by Dijkstra, Hoare and others. They called this the "correctness-by-construction" (CbC) style of software development and based it on weakest precondition semantics [8–12]. The approach should not be confused with other concepts that carry the same name, such as the correctness-by-construction (CbyC) promoted by Hall and Chapman [2]. Their CbyC is a software development process where formal modeling techniques and analyses such as the Z-notation are used for different development phases. Their goals are *to make it difficult to introduce defects in the first place, and to detect and remove any defects that do occur as early as possible after introduction* [13]. Another approach to correctness-by-construction is the Event-B framework [14] where automata-based system specifications are refined by provably correct transformation steps until an implementable program is obtained [15,16].

Several other approaches exist in which correct-by-construction systems are developed by synthesis or composition. Lamprecht et al. [17] present a synthesis-based approach to derive variants of a product family that are correct-by-construction by assembling existing building blocks with respect to a set of given constraints. Similarly, de Vink et al. [18] show how the CIF3 supervisory control tool allows to automatically synthesise a behavioral model of an SPL by starting from a feature model, component behaviour models associated with the features, and additional behavioural requirements in such a way that the resulting SPL model satisfies all feature-related constraints as well as all behavioral requirements. Kleijn et al. [19] study fundamental notions for the component-based development of correct-by-construction multi-component systems modeled as team automata. They provide precise conditions for the compatibility of components in systems of systems that (by construction) guarantee correct communications, free from message loss and deadlocks.

In contrast to these eponymous concepts, CbC starts by articulating a problem's pre-/postcondition specification and then derives a program from the specification in small, tractable refinement steps. Whenever a refinement step indicates that a loop structure is required, CbC requires that a suitable loop invariant and variant be stated before the body of the loop can be derived. As a result, CbC delivers not only a program that is 'correct by construction', but also the annotations required by PhV. The extent to which a CbC derived program can be guaranteed to be correct depends on the rigour with which the proof of each refinement step is undertaken. However, such rigour can be tedious and inefficient from a productivity perspective. To mitigate this problem, we argue for the lightweight application of CbC, followed by the application of PhV that can now direcly use the CbC-derived annotations that come along ‘for free’. Thus CbC should not be viewed as being in opposition to traditional PhV. Rather, CbC and PhV are complementary strategies for enhancing functional correctness.

To argue this position, we outline the CbC approach in the next section, emphasizing the development of loops. Section 3 then reflects on the relationship between CbC and PhV, indicating their relative strengths and weaknesses and emphasising, *inter alia*, loop termination. Section 4 briefly outlines our experiences on a case study in which PhV was applied to a CbC solution to an algorithmic problem and then attempted on a publicly available solution. The final section recommends combining CbC and PhV for the endeavour towards correct software and finishes with an outlook to future work.

2 Correctness-by-Construction

This section provides a short and necessarily superficial introduction to CbC. Here, we focus on CbC for loops, including invariants, variants and termination. We assume the reader has read [20, Sect. 2] for a brief introduction to the Dijkstra/Hoare style of CbC, and that the reader has a basic understanding of first order predicate logic (FOPL) formulae. A thorough introduction to CbC and related topics can be found in the ‘original’ books [8–10] (some of which are out of print or difficult to find) as well as [11] (available as a PDF from the author) and most recently [12]. We begin with a simple sorting algorithm before moving to a simplified graph closure algorithm, both of which are chosen to illustrate aspects of loop-design and termination.

2.1 A Simple Sorting Algorithm

CbC involves constructing a program (a.k.a. algorithm) from a specification using *refinement* steps. Given an algorithmic problem, CbC, thus, requires an articulation of the problem’s pre- and postcondition. For our purposes, such an articulation may be a pragmatic blend of natural language, FOPL, and diagrams. For example, if the problem is that of sorting a non-empty array A, it could be stated in a so-called pre-post formula:

$$\{A.len > 0\}\ S\ \{Sorted(A)\} \tag{1}$$

The foregoing is an assertion that states that if the length of array A is greater than 0 and some abstract command[1] S executes, then the command will terminate and the array will be sorted.

In this particular context, there is no compelling reason to provide a formal FOPL definition of what it means for an array to be sorted. Instead, we simply

[1] Dijkstra-speak for ‘program statement’.

assert the sortedness of array A by an undefined predicate $\mathit{Sorted}(A)$. In a similar spirit of lightweightness (seen in more places below), we also do not formalise that A contains the same elements before and after the algorithm executes—though now sorted.

Abstractly, the notation we use for pre-post specification (known as Hoare triples) looks like

$$\{P\}\ S\ \{Q\}$$

which specifies that 'assuming precondition P holds (is true), program statement (command) S *will* terminate and Q will then hold'. Refinement rules based on weakest precondition semantics allow for stepwise refinement of this pre-post specification. By convention, Dijkstra's Guarded Command Language (GCL) [11,12] is used to specify the programming commands that are embedded in the algorithmic specification. The refinement steps yield algorithmic specifications that embed increasingly detailed programming commands until we arrive at a specification that is sufficiently detailed to be translated into a programming language for compilation. Since GCL is an *imperative* pseudo-code, it can be translated to the method bodies of most object-oriented languages.

Returning to our need for a sorting algorithm, we initially appeal to some intuition and diagrams while designing a simple algorithm[2]. Since we do not know the length of array A *a priori*, we require *at least* one loop (a.k.a. a repetition command). This loop might move left-to-right through A using an index variable i, ensuring that everything strictly to the left of i is sorted, while the elements from i to the right may be unsorted. This 'ensurance' is encapsulated in a predicate called a *loop invariant*, and is graphically presented in Fig. 1. From the figure, we also note that when i goes off the right end of A (that is, $i = A.len$), we should *stop*, and since our invariant holds, A is sorted—our postcondition is established. Of course, we also require a plausible termination argument. Intuitively, we can see that, as long as our loop increments i in steps of 1 in each iteration (and no absurdities occur such as A spontaneously growing), we will go off the end of A and terminate. This is formalised with an integer-valued expression known as a *variant*, which is initially finite, can never be less than 0 and declines in each iteration, hence, it is bounded by 0. In our case, the

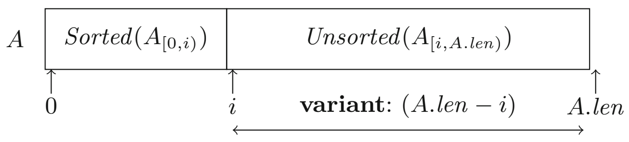

A
$\mathit{Sorted}(A_{[0,i)})$
$\mathit{Unsorted}(A_{[i,A.len)})$
0
i
variant: $(A.len - i)$
$A.len$

Fig. 1. Diagram of an invariant: that part of A strictly to the left of index i is sorted; subscript $[0, i)$ indicates that subrange from A strictly to the left of i. The variant is depicted as the 'distance for i to go' in A.

[2] We could, of course, apply ever deeper levels of intuition and arrive at the best known algorithms, but we limit our example here to the simplest sorting algorithms.

distance from i to $A.len$ fits the bill, and this is shown in the figure. In FOPL, the invariant I can be written as:

$$Sorted(A_{[0,i)}) \wedge (i \leq A.len)$$

Since it is relatively obvious, we do not bother to explicitly mention in I that $A_{[i,A.len)}$ is as yet unsorted. As mentioned before, when i goes off the right side ($i = A.len$), our invariant I implies $Sorted(A_{[0,A.len)})$, which is equivalent to our postcondition $Sorted(A)$.

We are now equipped to make two refinement steps rapidly. The first step takes us from (1) above and uses the 'sequence' (of commands) rule to give

$$\{A.len > 0\}\ S_1\ \{I\};\ S_2\ \{Sorted(A)\}$$

where we choose S_1 to do a minimal amount of work—simply set $i = 0$, which establishes I, since substituting $I[i := 0]$ gives

$$Sorted(A_{[0,0)}) \wedge (0 \leq A.len)$$

and the empty array segment $A_{[0,0)}$ is trivially sorted. We can now put the pieces together in the refinement step to introduce the loop[3], where the increment of i is already provided:

```
{ $A.len > 0$ }
$i : = 0$;
{ **invariant** $I$ and **variant** $A.len - i$ }
**do** $i \neq A.len \rightarrow$
    { $I \wedge \underbrace{i \neq A.len}_{\text{loop guard}}$ }
    $S_3$;
    $i : = i + 1$
    { $I \wedge$ **variant** $A.len - i$ has decreased and is non-negative }
**od**
{ $\underbrace{I \wedge \underbrace{\neg(i \neq A.len)}_{i=A.len}}_{Sorted(A)}$ }
```

Interestingly, at no point have we relied (in our correctness arguments) on the precondition $A.len > 0$. In fact, we could have omitted this restriction and accommodated empty arrays—the remainder of the algorithm would have been entirely correct. The precondition would then have been $\{A$is an array$\}$ or even more simply $\{$**true**$\}$. The first option makes explicit the type of A, and highlights

[3] Here, we have written the I in many places to emphasise where it *must* hold. In most algorithm presentations, it is only mentioned in the line preceding the loop, but the other proof obligations remain (in this case for S_3 to re-establish the invariant).

that it may not be ‘null’, must provide $A.len$ and be homogeneous; we have left out any formal discussion of types in this paper, though GCL contains types, declarations and scoping [11,12]. Correct algorithm behaviour in corner cases such as empty arrays are often overlooked by coders, or are so ‘intimidating’ that the precondition is then needlessly strengthened.

Clearly, at each loop iteration (increment of i), we will need to *do some work* to ensure our invariant still holds. Command S_3 must do something to integrate element A_i into the sorted portion $A_{[0,i)}$, and for this we have some algorithmic choices:

1. We can pairwise switch A_i with its left neighbour until it is in the correct sorted position—this *bubbling* action leading to bubble sort.
2. We can search $A_{[0,i)}$ to find the appropriate place j for A_i, then bump $A_{[j,i)}$ to the right by one position so A_i can fit at position j, in this case leading to insertion sort. To find the value of j
   (a) we can use linear search;
   (b) or, thanks to $\mathit{Sorted}(A_{[0,i)})$, we can use binary search

With all three of these possibilities, we would then refine S_3 into another loop—a step that is omitted here as it does not yield deeper insights into CbC. Lastly, as is shown in the algorithm, we note the variant decreases by 1 with every iteration and so the algorithm’s termination is assured[4].

We could have done this *algorithm derivation* much more formally, but this lightweight CbC is the essence of what we advocate, with the formalities being picked up as necessary by PhV as discussed in the coming sections.

2.2 A Simple Closure Algorithm

The previous section’s refinement to a sorting algorithm involved a variant which was relatively clear from the linear data-structure (array A). In this section, we work towards an algorithm with a more complex variant, and thus termination argument. One of the simplest closure-style problems is:

> Given a finite set N, a total function $f : N \longrightarrow N$ and an element $n_0 \in N$, compute the set $f^*(n_0) = \{f^k(n_0) : 0 \le k\}$ where $f^0(n_0) = n_0$ and $f^k(n_0) = f(f^{k-1}(n_0))$ for all $k > 0$.

This can be viewed as a problem over very simple directed graphs with nodes N, where f gives the successor of a node. Despite the simplicity, the graphs can take on a variety of forms, as illustrated in Fig. 2.
The specification of an algorithm solving the simple closure problem is:

$$\{N \text{ is finite} \wedge f : N \longrightarrow N \wedge n_0 \in N\}\ S\ \{D = f^*(n_0)\}$$

Intuitively, an algorithm computing $f^*(n_0)$ will calculate all $f^k(n_0)$ for increasing k, stopping when an already-seen element of N has been reached

[4] Again, this is barring absurdities such as the length of A changing dynamically, which is precisely the difficulty in parallel programs, in which this may indeed happen.

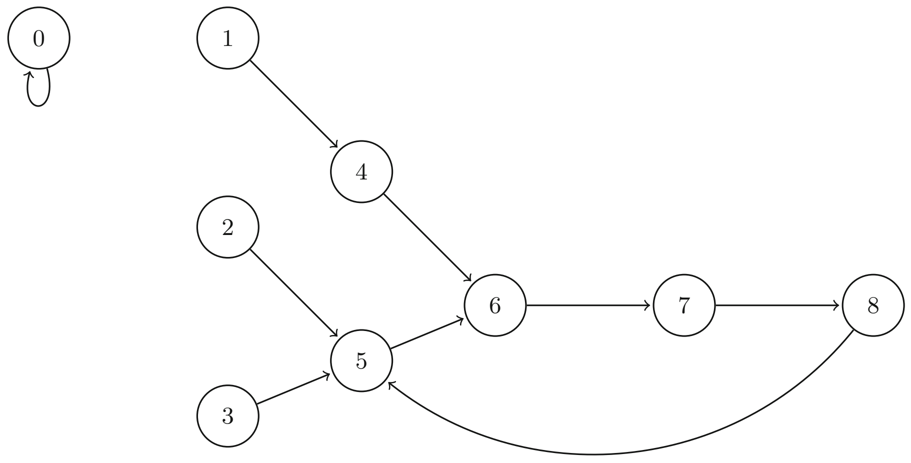


Fig. 2. Nodes representing N with arrows representing $f : N \longrightarrow N$. For example, $f^*(4) = \{4, 6, 7, 8, 5\}$

(variable D has already been presciently named for 'done'). To further refine, we introduce another set T for the 'to-do' elements; additionally, we introduce helper variable i to express the invariant J:

$$D = \{f^k(n_0) : k < i\} \wedge T = \{f^i(n_0)\}$$

We do not bother to specify trivialities such as $D \cap T \neq \emptyset$ and $D, T \subseteq N$, etc. This gives our first algorithm

```
{ $N$ is finite $\wedge f : N \longrightarrow N \wedge n_0 \in N$ }
$D, T, i :\, = \emptyset, \{n_0\}, 0$;
{ **invariant** $J$ }
**do** $T \neq \emptyset \rightarrow$
    { $J \wedge (T \neq \emptyset)$ } $S_0$ { $J$ }
**od**
{ $J \wedge (T = \emptyset)$ }
{ $D = f^*(n_0)$ }
```

As for our variant, we know that D cannot grow boundlessly since $D \subseteq N$ and N is finite. One possible variant is therefore $|N| - |D|$, though it is not particularly tight if we consider our example (in the caption of Fig. 2): $f^*(4) = \{4, 6, 7, 8, 5\}$ and at termination our variant is $9 - 5 = 4$, thus not reaching zero. Alternatively (as we do below), we can use the definition of f^* to give a tight variant $|f^*(n_0)| - |D|$. The latter variant of course *uses* f^* which is precisely what we are computing, and is probably therefore inappropriate for subsequent PhV; as a fall-back, the former, less tight variant may be used to still prove termination.

This gives our complete algorithm with the loop body refined to executable commands

```
$\{\ N$ is finite $\wedge f : N \longrightarrow N \wedge n_0 \in N\ \}$
$D, T, i :\ = \emptyset, \{n_0\}, 0;$
$\{$ **invariant** $J$ and **variant** $|f^*(n_0)| - |D|\ \}$
**do** $T \neq \emptyset \rightarrow$
    $\{\ J \wedge (T \neq \emptyset)\ \}$
    **let** $n$ such that $n \in T$;
    $D, T, i :\ = D \cup \{n\}, T - \{n\}, i + 1;$
    $\{\ D = \{f^k(n_0) : k < i\}\ \}$
    **if** $f(n) \notin D \rightarrow T :\ = T \cup \{f(n)\}$
    $[\!]\ \ f(n) \in D \rightarrow$ **skip**
    **fi**
    $\{\ T = \{f^i(n_0)\}\ \}$
    $\{\ J \wedge$ **variant** $|f^*(n_0)| - |D|$ has decreased and is non-negative $\}$
**od**
$\{\ J \wedge (T = \emptyset)\ \}$
$\{\ D = f^*(n_0)\ \}$
```

With this last closure algorithm (and the sorting algorithms in Sect. 2.1), we have exemplified CbC's ability to use small correctness-preserving refinement steps to arrive at algorithms which are elegant and immediately understandable, while simultaneously annotating the algorithm with assertions, invariants, and variants which directly and correctly arise from the refinements. With relatively little effort, the variants can then be used to prove termination. In the next section, we will see the further use of these artifacts in connecting CbC with PhV.

3 The Relationship Between CbC and PhV

Post-hoc program verification [4–7] assumes that a program to be verified is annotated with pre-/postcondition specifications for methods, and optionally class invariants in case of object-oriented programs. Additional annotations need to be provided to give the verification tools sufficient information in order to close proofs automatically. These additional annotations are, for instance, loop invariants and variants. Those annotations are classically expressed in FOPL formulae that characterise the program's variables, data structures and operations. Post-hoc program verification tools generally build on FOPL and corresponding provers and need to provide a calculus of the program semantics, i.e., how programs change the valuation of FOPL formulae.

We distinguish two general approaches for treating programs in program verification: (1) verification condition generation and (2) dynamic logic together with symbolic execution. In verification condition generation, the postcondition is transformed backwards through the program using a weakest precondition calculus. The effect of the program—i.e. the postcondition—is used to characterise the resulting weakest precondition formulae. What then needs to be shown is that the provided precondition logically implies this derived weakest precondition with respect to the given program code and postcondition. This proof goal is a FOPL formula. In the second approach, the program and its specification is translated into a dynamic logic formula, and the program within this formula is executed symbolically, capturing the program's effects in a symbolic state. After the program is completely evaluated and, thus, removed from the proof goal, the symbolic state can be evaluated for the remaining pre/postconditions such that a first-order proof goal remains.

3.1 The Case of CbC vs. PhV

Traditionally, the relationship between CbC and PhV is considered to be one of irreconcilable difference [21]. Usually, a picture of two opposite extremes is presented: PhV means arbitrary code is proved *ex post facto* to be correct with respect to its specifications; CbC means code that is rigorously evolved in a stepwise fashion that is guaranteed to be correct. In fact, it seems that there is a space in between these two extremes. For example, when applying PhV to the code, one could insist on certain constraints about how the code should be put together. For example, one could forbid the use of certain program constructs such as `repeat..until` commands, or require that it be expressed in a very simple language, or demand compliance with certain coding standards (such as MISRA-C, used in the automotive industry[5]).

For a meaningful combination of CbC and PhV, we propose the following workflow:

- Firstly, use CbC to derive an elegant algorithmic solution to the problem at hand, simultaneously providing pre/post-specifications and variant/invariant annotations. Here one should not fall into the trap of an 'analysis paralysis', by insisting that every detail has to be rigorously defined and proved. Instead, the emphasis should be on a pragmatic *lightweight* CbC derivation, in the sense described in Sect. 2. This, of course, increases the risk of error, but the risk can then be mitigated in the next step.
- Secondly, translate the CbC-derived program into the programming language that is required by the available PhV proof tool. It will also be necessary to translate the annotations into the logical notation syntax used by the tool. It might be necessary to provide the proof tool with additional annotations. For example, a classical CbC derivation might not be as concerned as a PhV tool with the explicit ranges of variables referenced in an invariant.
- Finally, use the prover tool to apply PhV to the translated CbC-derived program.

Ideally, assuming no errors were introduced in the CbC-based derivation or during the translation to the input language of the prover and enough additional annotations were provided, the proof should go through. Otherwise, iterative

[5] http://www.misra.org.uk.

debugging of the program and/or its annotations as well as their addition might be necessary. In the absence of any CbC tool support that embodies an integrated proof assistant, this workflow seems like an appropriate ‘marriage of convenience’ between CbC and PhV.

The workflow is based on the perception that CbC and PhV actually complement one another. It is designed to leverage their respective strengths and to mitigate their respective weaknesses as discussed below:

- A decided advantage of PhV is that it constructs a machine-checked proof that is correct, subject only to the correctness of the proof calculus and the correctness of the prover. However, a PhV weakness is that articulating the predicates to verify code that was developed in an *ad hoc* fashion with poor structure or no structure is non-trivial and sometimes not even possible. In contrast, CbC generally results in well-structured code, the code being a byproduct, so to speak, of articulating the specifications and annotations needed by PhV proof tools.
- CbC is concerned with correctness at the level of intuitive meaning. It deals with specifying the algorithmic solution to a problem and can tolerate lightweight, semi-formal or informal specifications provided they pragmatically capture the intuition of the developer. For example, in CbC it might be adequate to specify that an array A is sorted by simply writing down $Sorted(A)$. However, if CbC specifications and predicates are treated too informally, one risks errors. PhV can nicely fill this gap, allowing some informality of the CbC development, and then PhV checking with the invariants, variants, pre- and postconditions already worked out by the CbC effort. Of course, PhV tools need syntactically and semantically correct program statements and annotations to successfully complete a proof. So, for the above example, a PhV tool would need a detailed formal logical description what sortedness means, if this had not been provided by the lightweight CbC exercise. Even so, the PhV exercise starts off with a well-defined framework of what needs to be done, unlike what would have been the case if the PhV exercise was undertaken *ab initio*. So things become relatively easy on both sides, as it were.
- CbC allows the taxonomisation of algorithmic families [20]: At each refinement step, there may be several possible choices about precisely how to refine the specification, each choice leading to a different variant of the algorithm. These different refinement paths can then be used to guide taxonomisation of the various algorithms for the particular problem at hand, as we have seen for the sorting algorithms in Sect. 2. In the absence of such a structured refinement process to arrive at alternative algorithms solving the same problem, it becomes very difficult to discover characteristics that fundamentally distinguish such algorithms one from another, i.e., arbitrary or insignificant differences can become confused with fundamental differences. Hence, CbC allows the deep understanding of algorithmic families.

– One of the main drawbacks of CbC and, maybe, the biggest obstacle to its adoption, is the lack of tool support. If CbC had stronger tool support from the beginning and, hence, was more widely applied, CbC might have been prescribed in the development standards for safety-critical systems, instead of PhV. However, tool support for CbC strongly relies on the advances made for PhV program verification. Essentially, tool support for CbC would build on a FOPL prover and a calculus for capturing program semantics. Indeed, it would need to extend such provers with additional functionality, such as handling uninterpreted predicates and unknown program parts while a program is refined in CbC, in addition to different interaction and editing capabilities for the developer.

3.2 Termination-by-Construction

The literature on correctness distinguishes between total and partial correctness of a program. The notation introduced in Sect. 2, $\{P\}\ S\ \{Q\}$, is an assertion of *total* correctness. It evaluates to *True* if and only if the following holds:

If P is *True* and S executes, $\underline{\text{then} S}$ $\underline{\text{will terminate}}$ and Q will be *True*

By way of contrast $P\ \{S\}\ Q$ is an assertion of *partial* correctness[6]. It evaluates to *True* if and only if the following holds:

If P is *True* and S executes $\underline{\text{and}}$ $\underline{S}$ $\underline{\text{terminates, then}}$ Q will be *True*

The CbC approach to programming [8–12] is oriented towards deriving *totally* correct code *ab initio*. Not only does CbC require a variant, V, to be defined in lockstep with defining the loop's invariant, I. It also requires that the body of the loop, B, has to conform to the specification $\{I\}\ B\ \{I \wedge (0 \leq V < V_0)\}$ where V_0 is the value of V before B executes—i.e. it requires, by construction, that the variant strictly declines in each iteration of the loop towards a fixed lower bound (0 by convention). The CbC approach therefore seeks to avoid erroneously non-terminating code from the outset. One might say that CbC incorporates a Termination-by-Construction (TbC) approach to programming.

In contrast, PhV techniques operate on existing code. These techniques generally separate out the task of verifying that the code attains its postcondition (on condition that it terminates) from the task of verifying that the code indeed terminates. However, *partial correctness* is a weak concept in the sense that all specifications of non-terminating programs are *True* assertions[7]. This counter-intuitive observation focusses attention on the nature of non-terminating code, i.e. on whether it is intentionally or mistakenly non-terminating. There are of course, isolated instances of coded solutions to problems where the termination properties remain a matter of conjecture. One such example is the well-known

[6] There are alternative notational conventions in the literature for total and partial correctness.

[7] This is because both *False* $\implies$ *True* and *False* $\implies$ *False* evaluate to *True*,.

Collatz conjecture [22] that the algorithm generating the so-called hailstone sequence of numbers will always terminate.

If code is intentionally non-terminating, (e.g. an operating system that is driven by an infinite loop), then such code is *ipso facto* not focussed on attaining progressively a specific postcondition. Instead, such code typically requires that various interim postconditions should hold each time certain chunks of code within the body of the non-terminating loop complete. Of course, it might also be appropriate in such a scenario to ensure that certain globally invariant conditions are retained throughout the code—for example, the preservation of historical information in the event of an unanticipated hardware interrupt.

The point on which to focus is that postcondition semantics is only meaningful in those sections of the code that are intended to terminate. Our concern here is with such code and, in particular, with how such code should be constructed. There are a number of well-known traditional structured programming heuristics to improve the readability and maintainability of code in respect of its termination properties [8]. Examples include the avoidance of 'go to', 'break' and 'return' statements to exit loops. Such heuristics are oriented towards simplifying the task of understanding a loop's behaviour. There is a single easy-to-identify exit point of each loop. The condition for transiting through this exit point is easily located and clearly articulated, namely in the loop's condition. It goes without saying that TbC produced code complies with all these heuristics.

However, allegorical evidence suggests that these heuristics tend to be widely ignored, not only in private code, but also in industrial code and even in code intended for public inspection and use that is placed on open forums. Section 4 will give examples of such code.

It would be foolhardy to neglect tried and tested heuristics on the grounds that PhV tools are available to check for termination. We advocate, instead, for the disciplined TbC approach to code construction. A 'marriage of convenience' between CbC and PhV can be expected to benefit termination correctness in much the same way as it will enhance correctness in other areas of concern.

4 Case Study

This section reports on our experience in marrying CbC and PhV. It illustrates how PhV applied to ugly hacked-into-correctness code (often unashamedly made available on public forums) is difficult, if not impossible. This stands in contrast to applying PhV to clean, well-structured, easy-to-understand code for the same problem, as delivered by a CbC approach. As a simple example, we considered how the CbC approach would solve the *Partition* sub-algorithm used in the well-known Quicksort algorithm [23,24]. Assume that Quicksort is being applied to the array A. Recall that the purpose of *Partition* is to reorganise a sub-array $A_{[\ell,h+1)}$ into a lower section whose elements are less than or equal to a pivot element, say A_ℓ, all the elements in the remaining upper section then having elements greater than A_ℓ. *Partition* returns the boundary, say j, of these two subarrays. The upper diagram in Fig. 3 illustrates the postcondition of *Partition*,

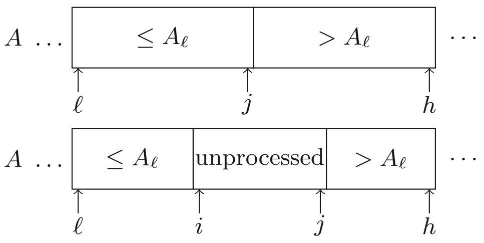


Fig. 3. Diagram of the postcondition and invariant used in *Partition*

```
proc  Partition(A, ℓ, h)
        { pre  $\equiv \ell < h$ }
        $i, j := \ell + 1, h$;
        { inv $\equiv \forall k \in [\ell, i) : (A_k \leq A_\ell) \wedge \forall r : (j, h] : (A_r > A_\ell)$ }
        { variant : $(j + 1 - i)$ }
        do  $(i \neq j + 1) \rightarrow$
             if  $(A_i \leq A_\ell) \rightarrow i := i + 1$
             ▯  $(A_j > A_\ell) \rightarrow j := j - 1$
             ▯  $(A_i > A_\ell) \wedge (A_j \leq A_\ell) \rightarrow A_i, A_j := A_j, A_i;\ i, j := i + 1, j - 1$
             fi
        od;
        { (inv $\wedge (i = j + 1)) \Rightarrow$ post }
        { post  $\equiv \forall k \in [\ell, j] : (A_k \leq A_\ell) \wedge \forall r : (j, h] : (A_r > A_\ell)$ }
        return $j$
corp
```

Fig. 4. CbC-derived version of *Partition*

and the lower diagram shows an interim state of the algorithm that relies on variable i and j to demarcate the subscript range of unprocessed elements in $A_{[\ell,h+1)}$.

A GCL version of *Partition* is given in Fig. 4. It was derived in a lightweight CbC fashion. As is customary, the pre- and postcondition, and the loop invariant used in the derivation, were included in the algorithm as FOPL assertions embedded between the various commands. Also left in comments is the integer expression representing the loop's variant. The flow of logic and correctness of the algorithm is clear. Variables i and j are initialised to establish the invariant of the loop. The loop's body consists of a single conditional command. This command specifies the conditions under which to increment i or decrement j. If neither of these conditions apply, then the third guarded command requires that A_i and A_j should be swapped, i should be incremented and j should be decremented.

The CbC rules ensure that all paths through the conditional statement result in the loop variant decreasing in each iteration and therefore in the loop eventually terminating. Additionally, the rules ensure that the invariant holds at the

```
algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i >= j then
            return j
        swap A[i] with A[j]
```

Fig. 5. Wikipedia version of *Partition* (June 27, 2016)

end of each loop iteration. Furthermore, it can easily be shown that upon termination of the loop, the invariant and the negation of the loop's condition imply the postcondition. In this sense, the algorithm's logic is transparent and readily verified as correct.

Consider, by way of contrast, an alternative rendition of the same algorithm as given in Wikipedia's entry for Quicksort given in Fig. 5. It is significantly more difficult to follow the flow of logic in this version of the algorithm, and thus to have confidence in its correctness. Here are some of the perceived problems with this code:

– The algorithm introduces an (arguably redundant) `pivot` variable
– There is no guiding principal about why `i` should be one less than `lo` rather than the same as or one more than `lo`. Similarly in regard to `j`. By way of contrast, in the CbC version initialisation of `i` and `j` is specifically aimed at establishing the invariant.
– Use of an infinite loop unnecessarily violates good coding standards. It requires an exit point and this is found at the second last statement. This imposes an additional intellectual effort to verify whether the condition of the if-statement (`i` >= `j`) makes sense in the context.
– The infinite loop embeds two successive `do..while` loops. These are inherently difficult to reason about, since each entails a first unconditional execution of the body followed by the evaluation of a condition. Evidently the intention of the first inner loop is to increment `i`, while the intention of the second inner loop is to decrement `j` as far as possible. Clearly, the logic required to verify that there is no off-by-one error in either of these loops is much more intricate than in the case of the CbC-based algorithm.
– After these two loops, the algorithm checks whether `i` >= `j` and terminates returning `j` if this is the case; otherwise it swaps `A[i]` and `A[j]`. Once more, it is non-trivial to become convinced that this exit condition does not entail an off-by-one error.

There is much allegorical evidence to support the claim that poorly structured code such as this is common both in industrial software and on open forums. As another example, consider the Java function, `edmondsKarp`, in Wikibooks that implements the Edmonds-Karp algorithm for computing maximal flow in a network[8]. Due to space limitations, the function will not be reproduced here. It can be seen that the function also issues a return from within an if-statement that is embedded in an infinite loop—as in the Wikipedia version of the *Partition*

```
public class Partition {
/*@ normal_behavior
 @ requires l < h;
  @ requires 0 <= l;
  @ requires h < A.length;
  @ ensures (\forall int k; l <= k && k <= \result;  A[k] <= A[l]);
  @ ensures (\forall int r; r > \result && r <= h ; A[r] > A[l]);
  @ assignable A[*];
  @*/
public static int partition(int[] A, int l, int h) {
        int i = l + 1;
        int j = h;
        int temp; // for swapping
        /*@ loop_invariant l < i && i <= j+1 && j <= h;
         @ loop_invariant (\forall int r; r > j && r <= h ; A[r] > A[l]);
         @ loop_invariant(\forall int k; l <= k && k < i;  A[k] <= A[l]);
         @ assignable A[*], i, j, temp;
         @ decreasing (j + 1 - i) ;
         @*/
        while (i != j + 1 ) {
            if (A[i] <= A[l]) { i = i + 1;}
                else if (A[j] > A[l]) {j = j - 1;}
                else if (A[i] > A[l] && A[j] <= A[l]) {
                    temp = A[i];
                    A[i] = A[j];
                    A[j] = temp;
                    i = i + 1;
                    j = j - 1;
                }
        }
        return j;
    }
}
```

Fig. 6. JAVA program with JML [25] annotations for CbC-derived *Partition* function used for PhV verification with KeY [4]

[8] https://en.wikibooks.org/wiki/Algorithm_Implementation/Graphs/Maximum_flow/Edmonds-Karp.

algorithm example above. Even worse, the code breaks out from a loop with the following skeletal structure:

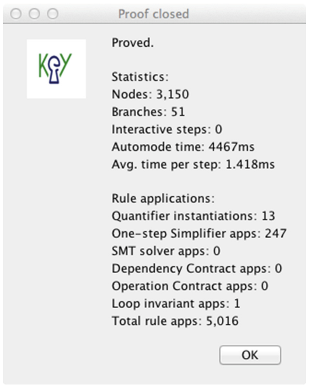
```
LOOP: while(C1){...for(C2){...if(C3){...}else{...break LOOP}}}
```

i.e. it breaks from the else-part of an if-statement in a for-loop that is embedded in a while loop!

That code such as this is routinely found in industrial software and unashamedly presented on public platforms ought to be concerning for software professionals at many levels, not least because it erodes the professional obligation of maintaining and verifying code.

To corroborate the claims that have been made above about the benefits of marrying CbC and PhV, and the difficulties in applying PhV to arbitrary code, an attempt was made to apply PhV to the two *Partition* versions given above. The KeY [4] PhV tool was used for this purpose. In each case, the code had to be translated into Java. This was an easy exercise, the only slight variation being that an additional variable, `temp`, was introduced to implement the swap. In the case of the CbC version, the FOPL comments were painlessly translated into the JML [25] annotation syntax required by the tool. In addition, it was necessary to indicate a lower bound on the variable ℓ, an upper bound on h as well as to indicate that array A is assignable. The resulting input code and tool output is reproduced in Figs. 6 and 7 affirming the algorithm to be correct.

Matters were considerably more complicated in the case of the Wikipedia code. A few trial traces by hand through the code seemed to deliver the correct answer, despite an uncomfortable intuition that the condition on the first inner loop should contain `<=` rather than `<`. To explore correctness more fully, an attempt was made to annotate the KeY (Java) version of the program. Pre- and postcondition annotations were not considered a problem since they would largely correspond with those used to derive the CbC-based algorithm. Additionally, `j-i` seemed like a reasonable variant for the infinite loop. However, other KeY-required annotations were not at all obvious.



```
Proved.

Statistics:
Nodes: 3,150
Branches: 51
Interactive steps: 0
Automode time: 4467ms
Avg. time per step: 1.418ms

Rule applications:
Quantifier instantiations: 13
One-step Simplifier apps: 247
SMT solver apps: 0
Dependency Contract apps: 0
Operation Contract apps: 0
Loop invariant apps: 1
Total rule apps: 5,016
```

Fig. 7. Screenshot of KeY [4] output on proof of the CbC-derived *Partition* function

We were not able to articulate meaningful invariants for the infinite loop and the two `do B while(C)` loops. It is probable that appropriate invariants will be found if the loops are transformed into semantically equivalent standardised formats (e.g. transform `do B while(C)` to `B; while(C) B`). However, such transformations violate the principle of carrying out PhV on code as-is. We therefore decided to abandon the effort of carrying out PhV on the code.

5 Conclusions

Many contemporary software systems have stringent functional correctness requirements. This paper has proposed a *lightweight* approach to CbC as a first step to meet such demands. For example, not all annotations used in a CbC-based program derivation need to be spelt out with full formal rigour. Similarly, some latitude may be allowed in accepting the correctness of certain refinement increments without carrying out detailed correctness proofs. In doing so, it is hoped that algorithmic solutions may be achieved more efficiently. However, it is also acknowledged that this could reduce the solution's effectiveness because the risk increases of accidentally introducing errors.

Combining lightweight CbC with PhV mitigates this risk. Moreover, the burden of formulating annotations for the PhV proof checker will be lightened by the availability of CbC-produced annotations, even if they have not been formally elaborated. In addition, we have also shown that CbC tends to produce a well-structured algorithmic solution — one that is generally far more amenable to PhV than code developed in an *ad hoc* (hacked) fashion. Hence, CbC and PhV should be seen as complementary strategies for enhancing functional correctness, and brought together in a 'marriage of convenience'.

This also means that CbC should be taught more widely than is currently the case, in training professional software developers. Candidates who have been subjected to the mental discipline CbC imposes (such as rigorously defining predicates and proving refinement steps) tend to have a greater awareness of corner cases to be considered and an appreciation of the value of structure and elegance in code. Beyond formal CbC training, though, *lightweight CbC* should be widely used, even in the presence of PhV verification tools, or if PhV is mandated by development standards for safety-critical software.

For future work, we propose to consider CbC approaches for programming models and languages other than sequential programs considered in this paper. CbC approaches should be considered for deriving algorithms in targeted domain-specific languages [26] that are then used to directly generate actual implementations. Additionally, CbC approaches for parallel programs have high potential to improve application correctness and enhance provability despite the complexity of parallelism. Finally, CbC tools in the form of structured editors that directly support the CbC style of code derivation — by, for example, carrying out automated proofs of each refinement step — would greatly advance the cause of professional software development.

Acknowledgments. This work was partially supported by the DFG (German Research Foundation) under grant SCHA1635/2-2, and by the NRF (South African National Research Foundation) under grants 81606 and 93063.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co. Inc., Boston (2000)
2. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. Softw. IEEE **19**(1), 18–25 (2002)
3. Beckert, B., Hähnle, R.: Reasoning and verification. IEEE Intell. Syst. **29**(1), 20–29 (2014)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
5. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Havelund, K., Holzmann, G.J., Joshi, R., Bobaru, M. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
6. Barnett, M., M. Leino, K.R., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
7. Filliâtre, J.-C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Upper Saddle River (1976)
9. Gries, D.: The Science of Programming. Springer, Heidelberg (1987)
10. Cohen, E.: Programming in the 1990s: An Introduction to the Calculation of Programs. Springer, Heidelberg (1990)
11. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, Upper Saddle River (1994)
12. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012)
13. Chapman, R.: Correctness by construction: a manifesto for high integrity software. In: Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software. SCS 2005, vol. 55, pp. 43–46(2006)
14. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
15. Méry, D., Monahan, R.: Transforming event B models into verified C# implementations. In: First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, 12–13 July 2013, pp. 57–73 (2013)
16. Cheng, Z., Mery, D., Monahan, R.: On two friends for getting correct programs - automatically translating event-B specifications to recursive algorithms in Rodin. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 821–838. Springer, Heidelberg (2016)
17. Lamprecht, A., Margaria, T., Schaefer, I., Steffen, B.: Synthesis-based variability control: correctness by construction. In: Formal Methods for Components and Objects, 10th International Symposium, pp. 69–88. Revised Selected Papers (2011)

18. ter Beek, M., Reniers, M., de Vink, E.: Supervisory controller synthesis for product lines using CIF 3. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 856–873. Springer, Heidelberg (2016)
19. ter Beek, M., Carmona, J., Kleijn, J.: Conditions for compatibility of components - the case of masters and slaves. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 784–805. Springer, Heidelberg (2016)
20. Cleophas, L., Kourie, D.G., Pieterse, V., Schaefer, I., Watson, B.W.: Correctness-by-construction $\wedge$ taxonomies $\Rightarrow$ deep comprehension of algorithm families. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 766–783. Springer, Heidelberg (2016)
21. ter Beek, M., Hähnle, R., Schaefer, I.: Correctness-by-construction and post-hoc verification - friends or foes? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 723–729. Springer, Heidelberg (2016)
22. Lagarias, J.C.: The 3x + 1 problem and its generalizations. IEEE Intell. Syst. **92**(1), 3–23 (1985)
23. Hoare, C.A.R.: Algorithm 64: quicksort. Commun. ACM **4**(7), 321 (1961)
24. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
25. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Commun. ACM **7**(3), 212–232 (2005)
26. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013). dslbook.org