Science of Computer Programming 97 (2015) 55–58

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Experience with correctness-by-construction

B.W. Watson [a], D.G. Kourie [b], L. Cleophas [b,*]

[a] Fastar Research Group, Information Science, Stellenbosch University, South Africa
[b] Fastar Research Group, Computer Science, Pretoria University, South Africa

HIGHLIGHTS

- We discuss the correctness-by-construction approach to software development.
- We discuss our experience with this approach in various algorithmic settings.
- We argue that its application to algorithmically complex system parts is worthwhile.

ARTICLE INFO

Article history:
Received 25 September 2013
Accepted 5 November 2013
Available online 19 November 2013

Keywords:
Correctness-by-construction
Refinement calculus
Algorithm derivation

ABSTRACT

We discuss the correctness-by-construction approach to software development, and our experience with this approach in various small to large scale algorithmic settings. We argue that although it is not realistic to apply the approach in developing every line of code of a software system, its pragmatic application to the algorithmically complex parts of such a system is worthwhile.

We have specially selected this topic, not only because of our own research interests, but also because correctness-by-construction lies close to Paul's own foundations in computing science, with (Prof. Dr.) Frans Kruseman Aretz. Being part of Paul's Ph.D. family tree, it has been a pleasure and an honour to share Paul's passion for programming language tools and to build upon his group's well-known advances in the field. Paul, we look forward to many more years of your research.

1. Introduction

Dijkstra's uncompromising differentiation between software engineering (SE) and his preferred style of software development are well known. He positioned himself as a prophetic voice crying out in the wilderness that only "the cruelty of really teaching computer science" [6] could solve the software crisis. SE, in contrast, was disparagingly characterised as "How to program if you cannot". This kind of polarising language has supported unfortunate stereotypes of industry developers hacking flakey code into production, while dilettante academics develop impractical theories about how code should be written. In this caricatured world, the former call themselves software engineers and the latter call their research "formal methods".

Here we argue against this view. We argue that the classical style of algorithm derivation espoused by Dijkstra, Hoare, Wirth and others—which has come to be known as correctness-by-construction (CbC)[1]—can be useful in the training and practice of SE. We do this by briefly presenting three case-studies in which CbC served a pragmatic role in our SE endeavours. The sample size is limited because of space considerations—many more examples could be given.

* Corresponding author.

[1] The earliest mention we are aware of is in [3].

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.024

2. Correctness-by-construction

By CbC we mean an approach to software construction that starts with an abstract specification of the problem at hand and that progresses in an ordered, step-wise fashion towards ever more refined (or concrete) specifications. At each step, rules that guarantee and thus prove correctness are followed. If applied strictly, then the final step delivers an algorithm that is guaranteed to be correct in the same sense that the proof of a mathematical theorem is correct. This requires that the semantics of notation in which the algorithm is articulated should be well-defined. Since Dijkstra's guarded command language (GCL) complies with this requirement and is also concise, it is commonly used as a notation. Several flavours of CbC have been advanced in texts such as [5,7,10] and [9].

The initial specification is in the form of pre- and postconditions, conventionally specified as first order predicate logic formulae. Refinement rules are based on the refinement calculus of [10] which relies, in turn, on the axiomatic semantics of [8] and also on the weakest precondition notions of [4]. In order to derive a loop, the approach pivotally depends on articulating a loop invariant as well as a loop variant.

It would be unrealistic, and indeed superfluous, to demand strict CbC adherence in routine software development. However, in our experience it has proved most beneficial when developing code that entails particularly complicated logic. Additionally, our experience suggests that a thorough educational grounding in CbC sharpens intuition, encourages an instinct to focus on invariants and reinforces a commitment to strive for *ab initio* correctness and elegance in the quest for quality software.

3. Selected case-studies

We offer three case-study examples to support our claims about the benefits of CbC. They illustrate how CbC has been deployed to derive useful, elegant and efficient algorithms. The first two examples illustrate our general contention: that by constructively deriving code from specifications, its overall structure and purpose tends to be more transparent and elegant than if unguided intellectual effort is brought to bear on the problem. In the interest of brevity, we do not discuss the derivations themselves nor the details of the application domain. Instead we focus on the overall flow of logic.

Example 1. The two versions below of an algorithm solve a computational geometry problem. The version on the left, *mal1*, was produced as part of a postgraduate research project. The version on the right, *mal2*, is CbC-derived, by refinement, starting out from the same pre- and postcondition, with some small refactorings for ease of exposition.[2]

```
proc mal1(...)
    F, j := φ, 1
    ; for h = 2 to n
        as ((h = n) cor ¬vis(j, h + 1)) →
            ℓ := j − 1
            ; do (ℓ ≠ 0 cor vis(ℓ, h)) →
                ℓ := ℓ − 1
            od
            ; F := F ∪ axline(ℓ + 1, h)
            ; j := h
        sa
    rof
corp
```

```
proc mal2(...)
    F, j, l, h := φ, 1, 1, 2
    ; do (j ≠ n) →
        c1(ℓ, h) := (h = n cor ¬vis(ℓ, h + 1))
        ; c2(ℓ, h) := (l = 1 cor ¬vis(ℓ − 1, h))
        ; if ¬c1(ℓ, h) → h := h + 1
        [] (c1(h, ℓ) ∧ ¬c2(ℓ, h)) → ℓ := ℓ − 1
        [] (c1(h, ℓ) ∧ c2(ℓ, h)) →
                F := F ∪ {axline(ℓ, h)}
                ; j, ℓ, h := h, h, h + 1
        fi
    od
corp
```

The original problem has an underlying symmetry that is directly expressed in *mal2*: when the parameterised predicates $c1$ and $c2$ both hold, then generate an axial line between ℓ and h and initiate the search for the next maximal axial line starting at $j = h$. If $c1$ does not hold then h should be incremented and if $c1$ does hold, but not $c2$, then ℓ should be decremented. This symmetry is submerged in a tangle of logic in *mal*1, requiring substantial reader effort to verify its correctness, especially at the boundary conditions. For example, the value of ℓ in *mal1* is not directly initialized to 1; the logic about when to increment h alone is split between the main loop and the negation of the condition of the **as** statement in the body; and the inner **do** loop disconcertingly decrements ℓ one unit past the value required by *axline* to generate the required axial line. As a result of this tangled logic, one is left with uncertainty about whether every boundary condition has been checked.

Example 2. In this example, *range1* is the GCL equivalent of a Python-implemented function that had been in use in a web development framework. The function reads chunks of a predetermined size from file (simulated here as array A) and

[2] In the algorithms F is the minimal set of maximal axial lines in a convex polygon chain. Polygons are numbered $1 \ldots n$. $vis(i, j)$ returns true iff i is visible from j. $axline(i, j)$ returns the axial line connecting i and j. **as .. sa** is a GCL extension representing a normal if-statement (i.e. in contrast to GCL's **if**, which aborts if none of the guards of the statement is true). **cand** and **cor** are conditional (short-circuit) and- and or-operators respectively.

processes what was read (simulated here by function call *yield*(*chunk*)). Reading starts and ends at parameterised positions s and e respectively. In conformance with the given specification, if $s < 0$ then reading starts at the start of A and if $e < 0$ then reading continues to the end of A. A CbC-derived version of this algorithm, *range2*, is given alongside *range1*. Its focus is a simple loop invariant stating that $A_{[s,c)}$ has been processed. This naturally uncovers three tasks to be addressed in turn. 1) Establish the loop invariant, taking the parameterised bounds into account. 2) Use a loop to read and process chunks of constant length as far as possible without exceeding the specified end-bound. 3) Read and process the data left over in the specified range. Because *range1* deals with all these concerns in the same loop, it has to rely on flags and conditional statements. In addition, in keeping with the C/C++ tradition, it treats *chunk* as both a boolean and an array variable. All of this results in unnecessarily complicated logic.

```
proc range1(s, e)
    {pre s, e ∈ [−1, A.len) ∧ (s ⩽ e)}
    c := 0;
    ; as (s < 0) → c := s sa
    ; done := false
    ; do (¬done) →
        ; chunk := read(A[c,chunk.len))
        if (¬chunk) → done := true
        [] (chunk) → c := c + chunk.len
            ; as ((e < 0) ∧ (c > e + 1)) →
                over := e − c + 1
                ; chunk := chunk[: over]
                ; done := true
            sa
        fi
        as (chunk) → yield(chunk) sa
    od
    {post A[s,e+1) has been processed}
corp
```

```
proc range2(s, e)
    {pre s, e ∈ [−1, A.len) ∧ (s ⩽ e)}
    as (s < 0) → s := 0 sa
    ; as (e < 0) → e := A.len − 1 sa
    ; c := s
    {invariant A[s,c) has been processed}
    {variant V ≡ (e + 1 − c)}
    ; do (c + chunk.len ⩽ e + 1) →
        ; chunk := read(A[c,c+chunk.len))
        ; yield(chunk)
        ; c := c + chunk.len
    od
    {invariant ∧ (c + chunk.len > e + 1)}
    ; as (c < e + 1) →
        chunk := read(A[c,e+1)); yield(chunk)
    sa
    {post A[s,e+1) has been processed}
corp
```

The third example shows how CbC can be used productively to classify existing algorithms that solve a specific problem, and to thereby stimulate ideas for alternative algorithms. The starting point is to find an algorithmic solution to the problem at hand that is sufficiently abstract to serve as a base from which all existing algorithms are subsequently derived in a CbC manner. These known algorithms will typically have different authors who each have their own presentation style, conventions and notational preferences. The algorithms may also rely on different data structures. As a result, it may require significant effort to filter all this diverse source information into a format that can be used for deriving each algorithm *ab initio* in a CbC fashion. Included in this effort will be a quest for domain-specific notation to express pre- and post-conditions and invariants of the various algorithms in a mutually compatible way. Each CbC derivation starts from the common base abstraction, but follows its own unique refinement path. These refinement paths provide the basis for a taxonomy showing how the various algorithms are related.

Example 3. To illustrate these ideas, consider the problem of finding the minimum acyclic deterministic finite automaton (MADFA) for a given language. In [12], the base algorithmic abstraction presented below was proposed for representing all known algorithms that find the MADFA for a given language $W \subset \Sigma^*$.

Here (Q, δ, s, F) (with Q the set of states, δ the transition function, s the start state, and F the final state set) is an acyclic deterministic finite automaton (ADFA) (not necessarily minimal) whose language is D. Initially D is the empty set. In each iteration of a loop w is moved from T (initialised to W) to D. An abstract ordering ($\leqslant$) of words is assumed for selecting the minimum w in each iteration. A call to an abstract function *add_word*(w) appropriately updates Q, δ and F while preserving the abstract invariant *Struct*(D). When the loop terminates, another abstract function *cleanup*() is invoked to ensure that the ADFA is converted into a MADFA.

[12] produced seven versions of *Struct*(D), each leading to a CbC-derived version of *add_word*(w) and *cleanup*(). Five of these versions were existing algorithms, and one of the remaining two was inspired as a result of articulating the underlying abstract algorithm.

There are now dozens of algorithms included in such CbC-derived taxonomies. (Examples can be found in [11] and [1].) [13] illustrate how these taxonomies are an ideal basis for object oriented toolkits of a given problem domain. This is because the taxonomies clearly expose the hierarchical and component-wise relationship that algorithms have to another, and their (GCL) pseudocode is easily translated into e.g. C++ or Java code. Such toolkits, in turn, provide excellent platforms for benchmarking the inter-related algorithms. This TAxonomy-BAsed Software COnstruction (TABASCO) approach is discussed in [2].

```
{ pre  W ⊂ Σ* }
s := create();
(Q, δ, s, F) := ({s}, ∅, s, ∅);
D, T := ∅, W;
{ invariant: Struct(D)
  variant: |T| }
do  T ≠ ∅ → ⟦ var  w : Σ*
                | let  w : w is any minimal element of T under ⩽;
              { Struct(D) }
                Q, δ, F : add_word(w);
              { Struct(D ∪ {w}) }
                D, T := D ∪ {w}, T − {w}
              { Struct(D) }
              ⟧
od;
{ Struct(W) }
  Q, δ, F : cleanup()
{ post  Min ∧ ℒ = W }
```

4. Conclusion

The examples above illustrate some of the ways in which CbC can assist in addressing complexity and efficiency in large scale software development—specifically when challenging algorithmic tasks are encountered. The reasons for fostering CbC-based skills as part of a good education in computer science are therefore pragmatic rather than ideological. Two extremes should be avoided: the casual dismissal of CbC as dilettante, irrelevant and impractical in the SE armoury; and the equally arrogant dismissal of conventional SE tools and approaches (the various software processes, testing strategies, etc.) as fraudulent. Perhaps the central future challenge is to render CbC more easily accessible to aspiring and practicing software engineers.

References

[1] L. Cleophas, Tree algorithms: Two taxonomies and a toolkit, Ph.D. thesis, Eindhoven University of Technology, the Netherlands, Apr. 2008.
[2] L. Cleophas, B.W. Watson, D.G. Kourie, A. Boake, S. Obiedkov, TABASCO: Using concept-based taxonomies in domain engineering, South African Comput. J. 37 (Dec. 2006) 30–40.
[3] E.W. Dijkstra, A constructive approach to the problem of program correctness, circulated privately, http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF, Aug. 1967.
[4] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (1975) 453–457.
[5] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, NJ, 1976.
[6] E.W. Dijkstra, On the cruelty of really teaching computer science, Commun. ACM 32 (12) (1989) 1414.
[7] D. Gries, The Science of Computer Programming, 2nd edition, Springer-Verlag, 1980.
[8] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.
[9] D.G. Kourie, B.W. Watson, The Correctness-by-Construction Approach to Programming, Springer, 2012.
[10] C. Morgan, Programming from specifications, Online: http://web2.comlab.ox.ac.uk/oucl/publications/books/PfS/, 1998.
[11] B.W. Watson, Taxonomies and toolkits of regular language algorithms, Ph.D. thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sep. 1995.
[12] B.W. Watson, Constructing minimal acyclic deterministic finite automata, Ph.D. thesis, Department of Computer Science, University of Pretoria, 2010.
[13] B.W. Watson, L. Cleophas, SPARE Parts: A C++ toolkit for String PAttern REcognition, Softw. Pract. Exp. 34 (7) (2003) 697–710.