Transformational Programming and the Derivation of Algorithms

Martin Ward and Hussein Zedan
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
martin@gkc.org.uk and zedan@dmu.ac.uk

Abstract. The transformational programming, method of algorithm derivation starts with a formal specification of the result to be achieved (which provides no indication of how the result is to be achieved), plus some informal ideas as to what techniques will be used in the implementation. The formal specification is then transformed into an implementation, by means of correctness-preserving refinement and transformation steps. The informal ideas are used to guide the selection of transformations to apply: since they only guide the selection of valid transformations, the ideas do not themselves have to be formalised. At any stage in the process, sub-specifications can be extracted and transformed separately. The main difference between this approach and the invariant based programming approach (and similar stepwise refinement methods) is that loops can be introduced and manipulated while maintaining program correctness and with no need to derive loop invariants. Another difference is that at every stage in the process we are working with a correct program: there is never any need for a separate "verification" step. These factors help to ensure that the method is capable of scaling up to the dvelopment of large and complex software systems.

1 Introduction

The *waterfall model* of software development sees progress as flowing steadily downwards (like a waterfall) through the following states:

1. Requirements Elicitation: analysing the problem domain and determinining from the users what the program is required to do;
2. Design: developing the overall structure of the program;
3. Implementation: writing source code to implement the design in a particular programming language;
4. Verification: running tests and debugging;
5. Maintenance: any modifications required after delivery to correct faults, improve performance, or adapt the product to a modified environment [1]

In theory, one proceeds from one phase to the next in a purely sequential manner. But in practice, at each stage in the process, information may be

uncovered which affects previous stages. For example, during implementation it may be determined that the design is unsuitable and needs to be changed, during debugging the program implementation will have to be changed to fix the bugs, and so on. So the process described in Figure 1 includes feedback loops from each stage to preceding stages.

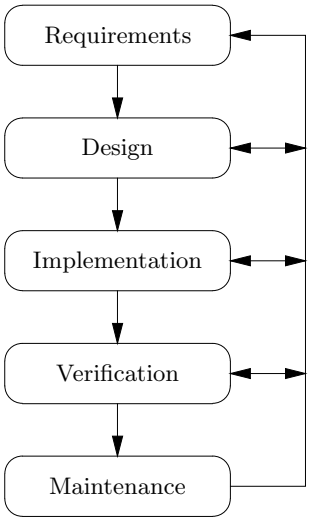

Fig. 1. The "waterfall model" of programming

It has long been recognised that while testing may increase our confidence in the correctness of a program, no amount of testing can prove that a program is correct. As Didjkstra said: "Program testing can be used to show the presence of bugs, but never to show their absence" [14]. To prove that a program is correct we need a precise mathematical specification which defines what the program is supposed to do, and a mathematical proof that the program satisfies the specification. In the case of a simple loop, the proof using the method of "loop invariants" takes the following steps:

1. Determine the loop termination condition;
2. Determine the loop body;
3. Determine a suitable loop invariant;
4. Prove that the loop invariant is preserved by the loop body;
5. Determine a variant function for the loop;
6. Prove that the variant function is reduced by the loop body (thereby proving termination of the loop);

7. Prove that the combination of the invariant plus the termination condition satisfies the specification for the loop.

This process is summarised in Figure 2. Loop invariants and postconditions can

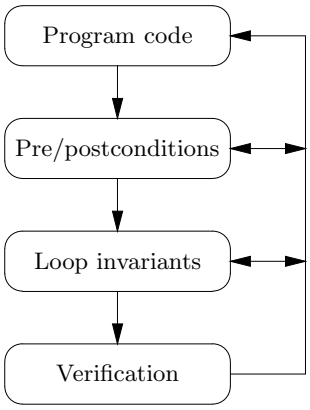

Fig. 2. The program verification process

be difficult to determine with sufficient precision, and computing verification conditions can be tedious and proving them can be difficult. Even with the aid of an automated proof assistant, there may still be several hundred remaining "proof obligations" to discharge (these are theorems which need to be proved in order to verify the correctness of the development) [7,18,22]. In addition, should the implementation happen to be incorrect (i.e. the program has a bug), then the attempt at a proof is doomed to failure.

Sennett in [23] indicates that for "real" sized programs it is impractical to discharge more than a tiny fraction of the proof obligations. He presents a case study of the development of a simple algorithm, for which the implementation of one function gave rise to over one hundred theorems which required proofs. Larger programs will require many more proofs. In practice, since few if any of these proofs will be rigorously carried out, what claims to be a formal method for program development turns out to be a formal notation for program specification, together with an *informal* development method.

An alternative to the *a posteriori* method, which was originally proposed by Dijkstra [12], is to control the process of program generation by constructing loop invariants in parallel with the construction of the code. This process is summarised in Figure 3. Combined with *stepwise refinement* [14,30], this approach is claimed to scale up to fairly large programs.

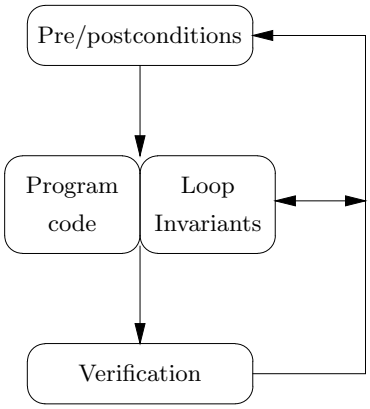

Fig. 3. Dijkstra's approach

A further refinement of this approach is to develop loop invariants *before* the code itself is written. The idea has been proposed from the late 70's by several researchers in different forms. Dijkstra's later work on programming [13] emphasises the development of a loop invariant as an important initial step before developing the body of the loop. Figure 4 summarises this approach. Notice that in Figures 2, 3 and 4, developing the loop invariant is moved to earlier and earlier phases in the process. Gries [16] takes up the idea that a proof of correctness and a program should be developed hand in hand. Back [2] presents a notation for writing invariant based programs, a method for finding invariants before writing code and methods for checking the correctness of invariant based program. He points out that the natural structure for the code may not be the same as the natural structure for the invariants and emphasises that the program should be structured around the invariants, so that they are as simple as possible.

In all the development methods we have seen so far, *verification* is the final step in development. Up until the point where verification has been completed, the programmer cannot be sure that the program is correct. Indeed, Back [2] makes it clear that the program under development does not have to terminate or be free from deadlocks, and that the initial invariant is usually both incomplete and partially wrong. He stresses that it is essential to carefully check the consistency of each transition when it is introduced.

In this paper we present a different method of programming, called *transformational programming* or *algorithm derivation*.

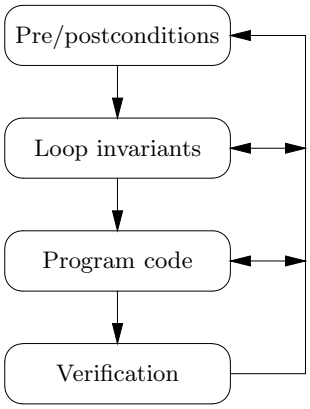

Fig. 4. Invariant based programming

2 Transformational Programming

The transformational programming method starts with a formal specification plus some informal ideas for implementation techniques which might be useful. The formal specification is *refined* into a complete program by applying a sequence of correctness-preseving refinement steps. The choice of which transformation to apply at each stage is guided by the implementation ideas. These ideas do not have to be formalised to any particular extent, since they are only used to select between different transformations. The correstness of the transformation guarantees that the transformed program is equivalent to the original. The method is summarised in Figure 5. Developing a program by stepwise transformation is an idea which dates back at least to the late seventies, starting with Burstall and Darlington's transformation work [10,11], the project CIP (Computer-aided Intuition-guided Programming) [3,4,5,6,9] and continuing more recently with the work of Morgal et al on the Refinement Calculus [19, 20,21] and the Laws of Programming [17]. However, the method presented here is very different from these. In the Refinement Calculus, the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. Morgan says: "The refinement law for iteration *relies on* capturing the potentially unbounded repetition in a single formula, the invariant", ([19] p. 60, our emphasis). So, in order to refine a statement to a loop, the developer still has to carry out all the steps 1–7 listed above for verifying the correctness of a loop.

In contrast with the refinement calculus, the method presented here does not require loop invariants. We have transformations to introduce, manipulate and remove loops which do not depend on the existence of loop invariants.

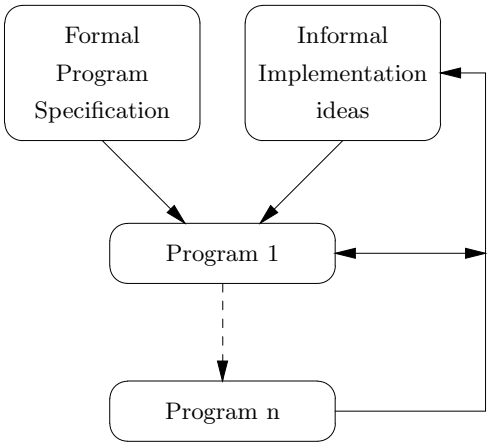

Fig. 5. Algorithm Derivation

Another key difference between Figure 5 and the other methods is that there is no Verification step. This is because at each stage in the derivation process we are working with a correct program. The program is always guaranteed to be equivalent to the original specification, because it was derived from the specification via a sequence of proven transformations and refinements.

Over the last twenty-five years we have developed a powerful wide-spectrum specification and programming language, called WSL, together with a large catalogue of proven program transformations and refinements which can be used in algorithm derivations and reverse engineering. The method has been applied to the derivation of many complex algorithms from specifications, including the Schorr-Waite graph marking algorithm [28], a hybrid sorting algorithm (an efficient combination of Quicksort and insertion sort) [26], various tree searching algorithms [27] and a program slicing transformation [29]. The latter example shows that a program transformation can be defined as a formal specification which can then be refined into an implementation of the transformation.

2.1 Outline of the Algorithm Derivation method

A typical algorithm derivation takes the following steps:

1. **Formal Specification:** Develop a formal specification of the program, in the form of a WSL specification statement. This defines precisely what the program is required to accomplish, without necessarily giving any indication as to how the task is to be accomplished. For example, a formal specification for the Quicksort algorithm for sorting the array $A[a\,..\,b]$ is the statement

SORT:

$$A[a\,..\,b] := A'[a\,..\,b].(\mathsf{sorted}(A'[a\,..\,b]) \ \wedge \ \mathsf{permutation}(A[a\,..\,b], A'[a\,..\,b]))$$

This states that the array is assigned a new value which is sorted and also a permutation of the original value. The formula $\mathsf{sorted}(A)$ is defined:

$$\forall 1 \leqslant i, j \leqslant \ell(A).\, i \leqslant j \Rightarrow A[i] \leqslant A[j]$$

while $\mathsf{permutation}(A, B)$ is defined to be true whenever the sequence B is a permutation of the elements of sequence A.

The *form* of the specification should mirror the real-world nature of the requirements. It is a matter of constructing suitable abstractions such that local changes to the requirements involve local changes to the specification.

The *notation* used for the specification should permit unambiguous expression of requirements and support rigorous analysis to uncover contradictions and omissions. It should then be straightforward to carry out a rigorous impact analysis of any changes.

The two most important characteristics of a specification notation are that it should permit problem-oriented abstractions to be expressed, and that it should have rigorous semantics so that specifications can be analysed for anomalies.

In [15], Dijkstra writes:"In this connection it might be worthwhile to point out that the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."

2. **Elaboration:** Elaborate the specification statement by taking out simple cases: for example, boundary values on the input or cases where there is no input data. These are applied by transforming the specification statement, typically using the Splitting a Tautology transformation followed by inserting assertions and then using the assertions to refine the appropriate copy of the specification to the trivial implementation. For Quicksort, an empty array or an array with a single element is already sorted, so SORT can be refined as **skip** in these cases:

$$\{a \geqslant b\};\ \mathsf{SORT} \ \approx \ \{a \geqslant b\};\ \mathbf{skip}$$

3. **Divide and Conquer:** The general case is usually tackled via some form of "divide and conquer" strategy: this is where the informal implementation ideas come into play to direct the selection of transformations. For Quicksort the informal idea consists of two steps: (a) Partition the array around a pivot element: so that all the elements less than the pivot go on one side and the larger elements go on the other side. (b) Then the two sub-arrays are sorted using copies of the original specification statement.

   At this point we still have a non-recursive program: so there are no induction proofs or invariants required for the transformations.
4. **Recursion Introduction:** The next step is to apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification.

5. **Recursion Removal:** We now have an executable implementation of the specification. If an iterative implementation is required, we can apply the Generic Recursion Removal Theorem (or an appropriate special case of the theorem) to produce an iterative program.
6. **Optimisation:** Apply further optimising transformations as required.

The method is *compositional* at several levels:

1. At any stage in the development, any part of the program can be worked on in isolation and the results "plugged back in" to the main program: this is because refinement in WSL satisfies the *replacement property* [25]
2. Different aspects of the development can often be handled separately: for example, correctness and efficiency;
3. At any stage in the process we may use data transformations to change the data representation and other transformations to introduce and remove "ghost variables" to convert abstract data structures to concrete data structures.

It should be noted that stages 1–3 involve analysing programs which contain no recursion or iteration. This makes the analysis particularly straightforward: for example, induction arguments are not needed. This is fortunate, as it is these stages which require the most input from the informal implementation ideas. Stages 4–6 involve standard transformations for recursion introduction, recursion removal and optimisation. As the derivation progresses, the transformations involved become more generic and less domain-specific. The techniques of *calculational programming* [8] may be relevant to these stages. In the later stages, the program will typically be getting larger, but the required transformations will become simpler and more susceptable to automation. At some point, an optimising compiler will take over and generate executable code, or the code will be directly executed by an interpreter, as appropriate.

An important advantage of the transformational derivation approach, over the various "code and verify" approaches is that it enables a separation of concerns between implementing the algorithm and applying various optimisation techniques. For example, in the derivation of the Schorr-Waite graph marking algorithm, the first stage involves deriving the following recursive graph marking algorithm from a specification for graph marking:

proc mark(x) $\equiv$
  $M := M \cup \{x\}$;
  if $L(x) \notin M$ **then** mark($L(x)$) **fi**;
  if $R(x) \notin M$ **then** mark($R(x)$) **fi.**

In the actual implementation, the values $L(x)$ and $R(x)$ are stored in arrays $l[x]$ and $r[x]$. The next stage involves using the "pointer switching" strategy to reduce the memory requirement for the recursive program. The recursive algorithm requires a stack to record the path back to the root. The central idea behind the algorithm devised by Schorr and Waite is that when we return from having marked the left subtree of a node we know what the value of $L(x)$ is

for the current node (since we just came from there). So while we are marking the left subtree, we can use the array element $l[x]$ to store something else—for instance a pointer to the node we will return to after having marked this node. Similarly, while we are marking the right subtree we can store this pointer in $r[x]$. The algorithm uses some additional storage for each node (denoted by the array $m[x]$) to record which subtrees of the current node have been marked.

In the final algorithm, the pointer switching strategy and the graph marking algorithm are deeply intertwined since the algorithm uses the same data structure for three different purposes: to store the original graph structure, to record the path from the current node to the root, and to record the current "state of play" at each node. The program is required to mark the graph without changing its structure, yet works by modifying that structure as it executes. This means that any proof of correctness must also demonstrate that all the pointers are restored on termination of the program. Any direct proof of the algorithm therefore has to prove that the graph is marked *and* that the pointers are restored simultaneously.

The transformational programming approach treats these two ideas separately in two stages of the derivation: we first derive a recursive graph marking algorithm, then apply the pointer switching strategy to show that various data items can be stored in the pointers, and the pointers restored at the end. Then remove the recursion, using this extra data to avoid the need for a stack. Proving that the pointers are restored is easy to do while the algorithm is in a recursive form: it becomes a much more challenging task when the algorithm is in its final iterative form.

3 Examples of Transformational Programming

3.1 String Comparison

Our first example illustrates multiple applications of the recursion introduction and recursion removal transformations. Recursion introduction does not have to be applied simultaneously to *every* copy of the specification: we can work on copies of the specification one (or more) at a time.

Given two character strings a and b, it required to determine whether they are equal "apart from blanks" (the space character being regarded as non-significant). We represent the strings as arrays of characters, with the special symbol end denoting the end of the string.

Define the function $\mathsf{strip}(s, i)$ to return the sequence of all non-space characters in s from the ith character to the end of the string:

$$\mathsf{strip}(s,i) = \begin{cases} \langle\rangle & \text{if } s[i] = \mathsf{end} \\ \mathsf{strip}(s, i+1) & \text{if } s[i] = \mathsf{space} \\ \langle s[i]\rangle +\!\!+ \mathsf{strip}(s, i+1) & \text{otherwise} \end{cases}$$

Formal Specification With this definition of strip our formal specification is:

$$\mathsf{COMP} =_{\mathrm{DF}} \textbf{if } \mathsf{strip}(a,1) = \mathsf{strip}(b,1) \textbf{ then } R := 1 \textbf{ else } R := 0 \textbf{ fi}$$

Informal Ideas Our informal idea is to step through both arrays a character at a time until we reach the end, or find a significant difference. This suggests generalising the specification to compare the strings from a given index onwards:

$$\mathsf{COMP}(i,j) =_{\mathrm{DF}} \textbf{if } \mathsf{strip}(a,i) = \mathsf{strip}(b,j) \textbf{ then } R := 1 \textbf{ else } R := 0 \textbf{ fi}$$

Program Derivation The obvious special cases to consider are the values of $a[i]$ and $b[j]$. First we consider the case where $a[i] = \mathsf{space}$:

if $a[i] = \mathsf{space}$ **then** $\mathsf{COMP}(i,j)$
else $\mathsf{COMP}(i,j)$ **fi**

By definition, if $a[i] = \mathsf{space}$ then $\mathsf{strip}(a,i) = \mathsf{strip}(a,i+1)$ so $\mathsf{COMP}(i,j) \approx \mathsf{COMP}(i+1,j)$. We have:

if $a[i] = \mathsf{space}$ **then** $\mathsf{COMP}(i+1,j)$
else $\mathsf{COMP}(i,j)$ **fi**

By the precondition for the program, there is an array element $a[i] = \mathsf{end}$ for some i. Let i' be the first such element. Then the variant function $i'-i$ is reduced before the first copy of the specification, but (obviously) not before the second copy. We can still apply Recursive_Implementation, provided we only apply it to the *first* copy of the specification:

proc comp $\equiv$
if $a[i] = \mathsf{space}$ **then** $i := i+1$; comp
else $\mathsf{COMP}(i,j)$ **fi**

This simple tail-recursion is transformed to a **while** loop:

while $a[i] = \mathsf{space}$ **do** $i := i+1$ **od**;
$\mathsf{COMP}(i,j)$

A similar argument for $b[j]$ produces:

while $a[i] = \mathsf{space}$ **do** $i := i+1$ **od**;
while $b[j] = \mathsf{space}$ **do** $j := j+1$ **od**;
$\mathsf{COMP}(i,j)$

Consideration of the cases where $a[i] = \mathsf{end}$ and/or $b[j] = \mathsf{end}$ gives:

while $a[i] = \mathsf{space}$ **do** $i := i+1$ **od**;
while $b[j] = \mathsf{space}$ **do** $j := j+1$ **od**;
if $a[i] = \mathsf{end} \wedge b[j] = \mathsf{end}$ **then** $R := 1$
elsif $a[i] \neq a[j]$ **then** $R := 0$
else $i := i+1;\ j := j+1;\ \mathsf{COMP}(i,j)$ **fi**

We can now apply Recursive_Implementation, and Recursion_Removal, to get the final iterative program:

do while $a[i] = \mathsf{space}$ **do** $i := i + 1$ **od**;
  while $b[j] = \mathsf{space}$ **do** $j := j + 1$ **od**;
  if $a[i] = \mathsf{end} \wedge b[j] = \mathsf{end}$ **then** $R := 1$; **exit**(1)
  elsif $a[i] \neq a[j]$ **then** $R := 0$; **exit**(1) **fi**;
  $i := i + 1$; $j := j + 1$ **od**

3.2 Quicksort

The Quicksort derivation uses the specification $\mathsf{SORT}(A, i, j)$ from Section 2.1. For the first step in the elaboration of the specification, we take out the case $i \geqslant j$. If $i \geqslant j$ then the array segment has at most one element, and is therefore already sorted. So a valid refinement of $\mathsf{SORT}(A, i, j)$ in this case is **skip**:

if $i < j$ **then** $\mathsf{SORT}(A, i, j)$ **fi**

The basic idea behind Quicksort is that the array segment is first partitioned around a pivot element such that each element on the left is smaller than each element on the right.

An improvement on the original Quicksort is to use *two* pivot elements, partitioning the array into three sub-arrays [31]. The left section contains all elements strictly less than the smaller pivot, the middle section contains all elements between the two pivots (inclusive), and the right section contains all the elements strictly larger than the larger pivot. This has been proved to require the same number of comparisons and 20% fewer swaps, in the general case, as compared to the traditional algorithm [31].

The specification $\mathsf{Partition}(A, i, j, p_1, p_2)$ is defined as:

$$\begin{aligned}\langle A, p_1, p_2\rangle := \langle A', p_1', p_2'\rangle.\big(&i \leqslant p_1' \leqslant j \wedge i \leqslant p_2' \leqslant j \\ &\wedge A'[i\,..\,p_1'-1] < A'[p_1'] \leqslant A'[p_1'+1\,..\,p_2'-1] \leqslant A'[p_2'] < A'[p_2'+1\,..\,j] \\ &\wedge \mathsf{Perm}(A, A', i, j)\big)\end{aligned}$$

where $\mathsf{Perm}(A, A', i, j)$ states that $A'[i\,..\,j]$ is a permutation of $A[i\,..\,j]$ and elsewhere A' is identical to A:

$$\begin{aligned}\ell(A) = \ell(A') \wedge \forall k.\,(1 \leqslant k < i \vee j < k \leqslant \ell(A) \Rightarrow A'[k] = A[k]) \\ \wedge \mathsf{Perm}(A, A', i, j)\end{aligned}$$

The three sub-arays $A[i\,..\,p_1-1]$, $A[p_1+1\,..\,p_2-1]$ and $A[p_2+1\,..\,j]$ can be sorted, using copies of SORT, in any order. Define a "pseudo-parallel" construct $(\mathbf{S}_1 \parallel \mathbf{S}_2)$ as:

$$\begin{aligned}(\mathbf{S}_1 \parallel \mathbf{S}_2) =_{\mathrm{DF}}\ &\textbf{if true} \to \mathbf{S}_1;\ \mathbf{S}_2 \\ &\Box\ \textbf{true} \to \mathbf{S}_2;\ \mathbf{S}_1\ \textbf{fi}\end{aligned}$$

Then, $\mathsf{SORT}(A, i, j)$ is equivalent to:

if $i < j$
  then $\mathsf{Partition}(A, i, j, p_1, p_2)$;
    $(\mathsf{SORT}(A, i, p_1 - 1) \parallel \mathsf{SORT}(A, p_1 + 1, p_2 - 1) \parallel \mathsf{SORT}(A, p_2 + 1, j))$ **fi**

The nondeterminacy in the $\parallel$ can later be refined in whatever way is most efficient.

Partition is refined as follows:

1. Select two distinct elements;
2. Swap the smaller element with $A[i]$ and the larger with $A[j]$;
3. Permute elements in the array $A[i+1\,..\,j-1]$ and assign to p_1 and p_2 such that:

$$A[i+1\,..\,p_1] < A[i] \leqslant A[p_1+1\,..\,p_2] \leqslant A[j] < A[p_2+1\,..\,j-1]$$

4. Swap $A[i]$ and $A[p_1]$ and swap $A[p_2]$ and $A[j]$.

Define:

$$\mathsf{Swap}(a, b) \;=_{\text{DF}}\; \langle A[a], A[b]\rangle := \langle A[b], A[a]\rangle$$

So the following program is an implementation of Partition:

$\langle p_1, p_2\rangle := \langle p_1', p_2'\rangle.(i \leqslant p_1 \leqslant j \;\wedge\; i \leqslant p_2 \leqslant j \;\wedge\; p_1 \neq p_2)$;
if $A[p_1] \leqslant A[p_2]$
  then $\mathsf{Swap}(i, p_1)$; $\mathsf{Swap}(j, p_2)$
  else $\mathsf{Swap}(j, p_1)$; $\mathsf{Swap}(i, p_2)$ **fi**;
var $\langle L := i+1, K := i+1, G := j-1\rangle$:
  $\mathsf{Part}(A, A[i], A[j], L, K, G)$;
  $p_1 := L - 1$;
  $p_2 := G + 1$;
  $\mathsf{Swap}(i, p_1)$; $\mathsf{Swap}(j, p_2)$ **end**

where $\mathsf{Part}(A, P_1, P_2, L, K, G)$ is defined as:

$\{P_1 \leqslant A[L\,..\,K-1] \leqslant P_2\}$
$\langle A, L, K, G\rangle := \langle A', L', K', G'\rangle.\big(A'[L\,..\,L'-1] < P_1 \leqslant A'[L'\,..\,K']$
$\qquad\qquad \leqslant P_2 < A'[G'+1\,..\,G]$
$\qquad\qquad \wedge\; L \leqslant L' \leqslant K' = G' + 1 \;\wedge\; L \leqslant G' \leqslant G$
$\qquad\qquad \wedge\; \mathsf{Perm}(A, A', L, G)\big)$

This specification is refined as follows:

if $K > G$
  then skip
elsif $A[K] < P_1$
    then $\mathsf{Swap}(L, K)$; $\mathsf{Part}(A, P_1, P_2, L+1, K+1, G)$
elsif $A[K] > P_2$
    then $\mathsf{Swap}(K, G)$; $\mathsf{Part}(A, P_1, P_2, L, K, G-1)$
    else $\mathsf{Part}(A, P_1, P_2, L, K+1, G)$ **fi**

After recursion introduction and recursion removal, we have:

while $K \leqslant G$ **do**
  if $A[K] < P_1$
    then Swap(L, K); $L := L + 1$; $K := K + 1$
  elsif $A[K] > P_2$
      then Swap(K, G); $G := G - 1$
      else $K := K + 1$ **fi od**

4 Conclusion

This paper presents a brief introduction to the Transformational Programming method of software development. The method starts with a formal specification of the result to be achievedtogether with some informal ideas as to what techniques will be used in the implementation. The formal specification is then transformed into an implementation, by means of correctness-preserving refinement and transformation steps. A key advantage of this approach is that loops can be introduced and manipulated while maintaining program correctness and with no need to derive loop invariants. Another advantage is that at every stage in the process we are working with a correct program: there is never any need for a separate "verification" step. These factors help to ensure that the method is capable of scaling up to the dvelopment of large and complex software systems.

In [24] Martyn Thomas writes:

> Software change is the *most important step* in the software lifecycle: most software costs far more after delivery than before (and most current "software maintenance" destroys more value than it preserves in your software assets).
> When requirements change, it is important to be able to make controlled changes to the specification. (In these circumstances, modifying software by going directly to the detailed design or code is vandalism). The specification therefore needs to be expressed in such a way that the nature, scope and impact of any change can be assessed and accommodated.

None of the "code and verify" development methods are particularly adept at handling changes to the specification. It is likely that the invariants will also change, and the whole development process will have to be repeated.

With transformational programming the prognosis is much better: with a properly-written specification (see Section 2.1), a small change to the requirements is likely to result in a small change to the specification. Any informal implementation ideas may still be valid: in which case, the derivation process can repeat many of the steps from the original derivation. This is because the implementation ideas are used to select the sequence of transformations to be applied: if the ideas are still valid then it is likely that the sequence is still valid and can be applied to the modified specification with only minimal changes. This process can be streamlined even further with the aid of suitable tool support, such as the FermaT Transformation System.

References

[1] ISO JTC 1/SC 7, "Software Engineering – Software Life Cycle Processes – Maintenance," ISO/IEC 14764:2006, 2006.

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

[3] F. L. Bauer, "Program Development By Stepwise Transformations—the Project CIP," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 237–266.

[4] F. L. Bauer, R. Berghammer, et. al. & The CIP Language Group, *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, Lect. Notes in Comp. Sci. #183, Springer-Verlag, New York–Heidelberg–Berlin, 1985.

[5] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 #2 (Feb., 1989).

[6] F. L. Bauer & The CIP System Group, *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, Lect. Notes in Comp. Sci. #292, Springer-Verlag, New York–Heidelberg–Berlin, 1987.

[7] Juan C. Bicarregui & Brian M. Matthews, "Proof and Refutation in Formal Software Development ," *In 3rd Irish Workshop on Formal Software Development* (July, 1999).

[8] Richard Bird & Oege de Moor, *The Algebra of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[9] M. Broy, "Algebraic Methods for Program Construction: the Project CIP," in *Program Transformation and Programming Environments* Report on a Workshop directed by F. L. Bauer and H. Remus, P. Pepper, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1984, 199–222.

[10] R. M. Burstall & J. A. Darlington, "A Transformation System for Developing Recursive Programs," *J. Assoc. Comput. Mach.* 24 #1 (Jan., 1977), 44–67.

[11] J. Darlington, "A Synthesis of Several Sort Programs," *Acta Informatica* 11 #1 (1978), 1–30.

[12] E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness.," Technische Hogeschool Eindhoven, EWD209, http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF.

[13] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[14] E. W. Dijkstra, "Notes On Structured Programming," Technische Hogeschool Eindhoven, EWD249, Apr., 1970, http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[15] E. W. Dijkstra, "The Humble Programmer," *Comm. ACM* 15 #10 (Oct., 1972), 859–866.

[16] David Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.

[17] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 #8 (Aug., 1987), 672–686.

[18] C. B. Jones, K. D. Jones, P. A. Lindsay & R. Moore, *mural: A Formal Development Support System*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[19] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[20] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[21] C. C. Morgan & T. Vickers, *On the Refinement Calculus*, Springer-Verlag, New York–Heidelberg–Berlin, 1993.

[22] M. Neilson, K. Havelund, K. R. Wagner & E. Saaman, "The RAISE Language, Method and Tools," *Formal Aspects of Computing* 1 (1989), 85–114 .

[23] C. T. Sennett, "Using Refinement to Convince: Lessons Learned from a Case Study," *Refinement Workshop, 8th–11th January, Hursley Park, Winchester* (Jan., 1990).

[24] Martyn Thomas, "The Modest Software Engineer," *The Sixth International Symposium on Autonomous Decentralized Systems, ISADS* (2003).

[25] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/thesis⟩.

[26] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/sorting-t.ps.gz⟩.

[27] M. Ward, "Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension," *Comput. J.* 42 #8 (1999), 650–673, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/recursion-t.ps.gz⟩ doi:10.1093/comjnl/42.8.650.

[28] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 #9 (Sept., 1996), 665–686, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/sw-alg.ps.gz⟩ doi:doi.ieeecomputersociety.org/10.1109/32.541437.

[29] Martin Ward & Hussein Zedan, "Deriving a Slicing Algorithm via FermaT Transformations," *IEEE Trans. Software Eng.*, IEEE computer Society Digital Library (Jan., 2010), ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/derivation2-a4-t.pdf⟩ doi:doi.ieeecomputersociety.org/10.1109/TSE.2010.13.

[30] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14 #4 (1971), 221–227.

[31] Vladimir Yaroslavskiy, "Dual-Pivot Quicksort," *Research Disclosure* RD539015 (Sept., 2009), http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf.