How to Produce Correct Software – An Introduction to Formal Specification and Program Development by Transformations

E. A. BOITEN, H. A. PARTSCH, D. TUIJNMAN AND N. VÖLKER*

University of Nijmegen, Department of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

The task of software production is to build software systems which are to fulfil certain requirements. For years the approach has been to build up by trial and error a program which, having satisfied carefully prepared test data, offers a plausible solution to the problem. But is it correct? Even for toy examples this is not obvious. In particular, it is often not even clear whether the original problem has been properly understood. The reason for this dilemma is that the transition from the informal problem statement to the final program is too big to be intellectually manageable. To master these problems, we advocate a software development method where the whole process is split into smaller steps by introducing formal specifications for (parts of) the problem and then stepwisely deriving efficient programs by correctness-preserving transformations.

Received June 1992

1. INTRODUCTION

The task of software production is to build software systems which are to fulfil certain, often vague, requirements. As a little example, which will accompany us in the remainder, consider the following problem:

Airports have electronically controlled boards to inform passengers of departure times of flights. The entries on the board are linearly ordered according to the (scheduled) departure times. Whenever a plane leaves, the respective entry on the board is eliminated and substituted by a blank entry ('gap'). Since not all planes leave on time, gaps usually will not only appear at the beginning of the board, but can be scattered all over it. After a certain period of time, usually the board is 'updated' by shifting all meaningful entries to the beginning of the board (without destroying the previous ordering) and thus accumulating all gaps at the rear. A request is received to develop a program for this update operation.

A programmer, confronted with this problem, might come up with the following program, written in a Pascal-like notation:

```
function update (t:board):board;
  begin
    var vi, vk:nat; var vt:board;
    vi := 1; vt := t;
    for vk from 1 upto |t| do
      if vt[vk] is gap
        then skip
        else vt[vi] := vt[vt]; vt[vk] := gap; vi := vi+1
      endif
    enddo;
    vt
  end
```

At first glance, this little program seems to be a plausible solution to the problem. But is it correct? Even for such a toy example, this is not obvious. In particular, it is not even clear whether the original problem has been properly understood. The reason for this dilemma is that the transition from the informal problem statement to the final program is too big to be intellectually manageable. To master these problems, we advocate a software development method where the whole process is split into smaller steps by introducing formal specifications for (parts of) the problem and then stepwisely deriving efficient programs by correctness-preserving transformations. Its essence is illustrated by Fig. 1.

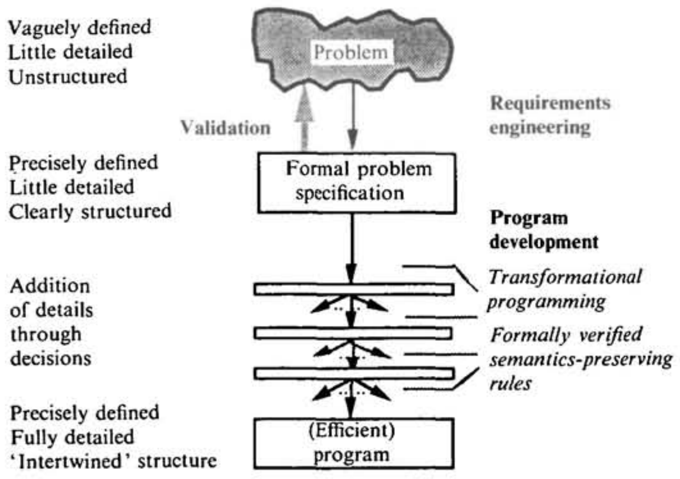

Figure 1.

From Fig. 1 it is obvious that making a formal specification is an essential first step for our approach. This may seem like extra work, but the formulation of such a specification forces the client to clarify his/her wishes at an early stage, thus decreasing the chance of futile work.[37] The transformation phase consists of the application of correctness-preserving transformations; thus the final result satisfies the formal specification by construction. Furthermore, the calculational nature of the transformations suggests building systems which support the transformation process.

The use of transformational programming is somewhat reminiscent of the use of mathematics in engineering. Just as few people would employ formal calculations for solving trivial problems, transformational programming

* This research has been carried out within the NFI-project STOP (Specification and Transformation Of Programs) and partially sponsored by NWO (Netherlands Organization for Scientific Research) under grant NFI-FW3315.

should not be seen as aimed at the derivation of simple, 'throwaway' programs. Instead, the particular strength of the method lies in deriving programs which are either very sophisticated or have to be 100 percent correct.

An additional benefit of transformational programming lies in the fact that many transformation rules are formalisations of programming knowledge. This can be helpful for a better understanding of programming in general and, in particular, should lead to more insight into the (partial) automation of programming. A further aspect is the possibility of reuse, i.e. parts of derivations can be used in the derivation of programs for related problems. This makes the approach not only useful for the development but also for the maintenance stage of the software life cycle.

The basic idea of transformational programming is not new; for 15 years research has been going on. By now, the method is well established in academic circles (cf. e.g. Ref. 27 for a comprehensive treatment of the subject), and has ripened enough to make it feasible for use in real-world applications.

2. FORMAL SPECIFICATIONS

An initial formal specification must give a clear and precise description of what the program-to-build should produce. The formalisation of an informally stated problem usually entails the formulation and use of high-level, abstract concepts. Therefore, a specification language has to be sufficiently rich to express these adequately. Below we present some specification constructs, such as algebraic data types or descriptive constructs, which are better suited for this purpose than those of conventional imperative or functional languages.

2.1. Algebraic specification of data structures

An algebraic specification gives a common description of data structures and the (basic) operations on these in an implementation-independent way. One only specifies the desired properties of these operations, without giving an operational description. We may consider a thus specified abstract data type as a 'black box', of which we can only observe the outside behaviour of the operations.

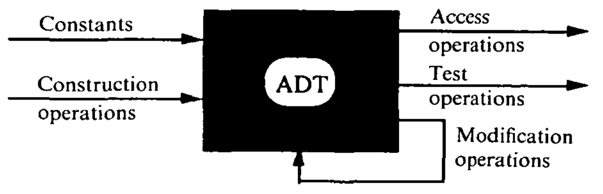

Figure 2.

Basically, an abstract data type is determined by its operations. As can be seen from the above picture, these fall conceptually into two classes. On the one hand, constants and constructors build 'new' objects. On the other hand, test and access functions extract information from an abstract object, and modification functions turn one object, and some primitive information, into another object.

The interrelations between the various operations are defined by (conditional) equations between terms, built from the constants and operations as described above. Each term represents an abstract object; an equation specifies that two terms represent the same object. This forms the essence of the semantics of algebraic specifications.

This way of specifying abstract data types gives great flexibility with respect to implementation: any concrete data structure, with appropriate operations on it, that fulfils the stated properties can be used. Moreover, the characteristic properties are fixed, and can be used in transforming a program – irrespective of what implementation will be chosen for the algebraic specification.

For formalising the essential data structure in our running example, the flight board, different abstract concepts might be appropriate. One possibility is to view it as a sequence of entries. Disregarding the nature of the particular entries, we first concentrate on a specification of the concept 'sequence'.

A sequence of objects (of arbitrary element type) is either empty (constant ε), or it is obtained by adding an object 'before' another sequence (constructor >+). In the latter case, the sequence can be decomposed into its first element and the rest of the sequence. As these two operations are partial, a function to test whether a sequence is empty (*isempty*) is also needed.

Formally, this yields the following specification, written in a Pascal-like notation. The header declares the name of the type and its parameters (here: an arbitrary sort m); the **exports** section lists those identifiers which are visible outside the type definition. Then follow the declarations of sorts, constants and operations; partial operations (such as *first* or *rest*) are provided with assertions restricting their domains. Lastly, the properties (laws), which were motivated informally above, are specified.

```
abstracttype SEQU = (sort m):
  exports sequ, ε, isempty, first, rest, _>+_;
  sort sequ,
  ε: sequ,                                          {empty sequence}
  function isempty (sequ): boolean,
                                               {test on emptiness}
  function first (s: sequ || ¬ isempty(s)): m,
                                                   {first element}
  function rest (s: sequ || ¬ isempty(s)): sequ,
                                       {all without first element}
  function _>+_ (m; sequ): sequ;
                                          {addition of an element}
  laws x: m; s: sequ ||
    isempty(ε) = true,
    isempty(x>+s) = false,
    first(x>+s) = x,
    rest(x >+s) = s
endabstracttype
```

The abstract notion of sequences thus specified can be implemented in various ways. One-way linked lists are an obvious possibility, but sometimes doubly linked lists may prove to be more useful; and when the sequences are known to be limited in length, arrays also can be considered. These are but a few of the alternatives.

Like functions, parameterised specifications, such as SEQU, can be instantiated with concrete arguments. For instance, to have sequences of natural numbers at one's disposal, one declares

type natsequ = SEQU(nat)

which gives a sort natsequ and the operations and properties of SEQU (with all occurrences of sort m replaced by nat). Instantiation facilitates the extension of an abstract type by new functions or extra properties, as shown in the example of indexed sequences below.

The exported operations of a type can be made available in another type by using the **based on**-construct. This enables the hierarchical construction of data types, and thus modularisation of the overall problem specification. As a practically important consequence, implementation of one type is possible independently of the choice of implementation for other, more primitive types.

Sequences as specified above only allow sequential access from left to right. Sometimes, however, direct access to arbitrary elements of a sequence is desirable. Using instantiation and hierarchical construction via **based on**, we can extend the specification of sequences with operations for indexed access:

```
abstracttype INDSEQU = (sort m):
  exports sequ, ε, isempty, first, rest, _>+_,
          |_|, _[_], _[_:_];
  based on NAT;
  type sequ = SEQU(m);
  function |_| (sequ):nat,
                                              {length}
  function _[_] (s:sequ; i:nat || 1 ⩽ i ⩽ |s|):m;
                                     {indexed access}
  function _[_:_] (s:sequ; i:nat, j:nat ||
                   1 ⩽ i ⩽ |s| ∧ 1 ⩽ j ⩽ |s|):m;
                                           {'slicing'}
  laws x:m; s:sequ; i:nat ||
    |ε| = 0,
    |x >+ s| = 1 + |s|,
    |s| ⩾ 1 ⇒ s[1] = first(s),
    i > 1 ⇒ s[i] = (rest(s))[i − 1],
    i > j ⇒ s[i:j] = ε,
    j ⩾ i ⇒ s[i:j] = s[i] >+ s [i + 1:j]
endabstracttype
```

Returning to our running example, we now may specify a flight table as an indexed sequence of elements:

type board = INDSEQU(elem)

where an element is either a 'real' entry or a gap, formally:

type elem = entry | gap.

Of course, due to space limitations, only the flavour of algebraic specifications can be given. For a more elaborate treatment, the reader is referred to (e.g.) Refs 4, 16, 17 and 41.

2.2 Descriptive specifications

The operations provided by an abstract data type form the basis for specifying algorithms. Just as algebraic specifications only define what result an operation has, and not how it is implemented, for an initial formal specification of an algorithm one is merely interested in describing its result, rather than how it is calculated. Therefore more abstract concepts are needed than purely imperative or functional constructs.

For predicates, for instance, quantifiers often form a useful means of describing a problem. Assuming an equality = on the element type, equality of two sequences can be specified as

```
function _=_ (s, t:sequ):boolean;
  |s| = |t| Δ ∀i:nat || (1 ⩽ i ⩽ |s|) ⤇ s[i] = t[i]
```

where the (sequential) conjunction Δ and implication ⤇ ensure evaluation from left to right in order to cope with partiality. Another example is testing whether all gap-entries on a board have been shifted to the rear, which is specified by

```
function normalised (t:board):bool;
  ∃s, s':board || t = s ++ s' ∧ no-gaps(s) ∧ all-gaps(s')
```

with ++, *no-gaps* and *all-gaps* as defined below.

In the same manner, a function result of a type other than boolean can be characterised by a predicate that gives the relation between arguments and result. For instance, an updated board has to have the same non-gap entries, in the same order, as the original board, and all gaps shifted to the rear:

```
function update (t:board):board;
  some s:board || |t| = |s| ∧ same-entries(s, t) ∧
                  normalised(s)
```

The operator **some** formalises a (non-deterministic) choice among all objects satisfying the given predicate. Thus *update* is a non-deterministic 'function', i.e. not a function in the mathematical sense. If a function is known to be determinate, this can be made explicit by using **that** instead of **some** – e.g. for joining two sequences:

```
function _++_ (s, t:sequ):sequ;
  that u:sequ || |u| = |s|+|t| Δ (u[1:|s|] = s ∧ u[|s| + 1:|u|] = t)
```

2.3 Applicative-functional specification

Of course, specifications can also be written with the usual constructs of functional languages: conditional expressions, as well as definition and application of functions, including recursive application. As functional languages are by now well established,[6, 19] we shall not go further into this subject, but restrict ourselves to some examples.

Appending an element at the end of a sequence can easily be specified using the join-function:

```
function _+<_ (s:sequ; x:m):sequ;
  s ++ (x >+ ε)
```

A filter function that yields the subsequence of all elements which fulfil some predicate is given by:

```
function filter (s:sequ; p:function (m):bool):sequ;
  if isempty(s)   then ε
  elsf p(first(s)) then first(s) > +filter(rest(s), p)
                   else filter(rest(s), p) endif
```

With this function, we can easily specify the functions *no-gaps*, *all-gaps* and *same-entries* mentioned above (where _is gap tests whether an object of type elem is of type gap):

```
function all-gaps (t:board):bool;
  filter(t, _is gap) = t;
function no-gaps (t:board):bool;
  filter(t, ¬(_is gap)) = t;
```

```
function same-entries (s, t:board):bool;
  filter(s, ¬(_is gap)) = filter(t, ¬(_is gap))
```

Moreover, functional constructs can be mixed freely with descriptive constructs.

2.4 Imperative constructs

In practice, most programs are written in imperative languages like C and Pascal. Therefore, we also allow imperative programs, but recommend their use mainly for the final goal of a program derivation. The essential part of the work should be done at the functional level, since functional languages have been shown to be more amenable to formal manipulation.

3. TRANSFORMATIONAL PROGRAMMING

The aim of a transformational development can vary a lot depending on the specification and also on non-functional requirements such as the degree of efficiency required, the kind of language in which the specifications/programs are stated and the target computer architecture. In general the following basic tasks can be distinguished:

- development of algorithmic solutions from algebraic or descriptive specifications;
- optimisation of programs with respect to their control structure;
- efficient implementation of data structures;
- adaptation of programs to different architectures.

Most transformation rules assert the equivalence between two pieces of program, denoted by $\equiv$, possibly under some conditions. This means that, in the context of a larger program, either piece may be replaced by the other without changing the meaning of the program. There are also rules which allow us, in one direction, to refine the semantics of a program, i.e. to make/restrict a choice out of several options left open in a specification.

Transformation rules can be divided into several groups depending on the constructs occurring within them and on their degree of complexity. Typical basic rules are the laws and derived properties of the underlying data types like the commutativity of multiplication, or the rule

$$isempty(s) \equiv |s| = 0$$

which holds in the type INDSEQU defined above. Further basic rules deal with the introduction, elimination and interchangeability of the constructs of the specification language, like the distributivity of function application over a conditional expression:

$$f(\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ endif}) \equiv \textbf{if } B \textbf{ then } f(E_1) \textbf{ else } f(E_2) \textbf{ endif}$$

Of particular importance are the elementary rules about functions, like the instantiation of a function with actual parameters, called 'unfolding', and the reverse 'folding' operation, which replaces an expression by a suitable function application.[11] In this context we must also mention 'generalisation': the introduction of a new function by adding extra parameters to an already existing one. The combination of these rules on functions, together with the other basic rules, forms a quite powerful calculus which is sufficient to solve the first two tasks mentioned above.

The set of transformation rules does of course not have to be fixed. Instead, the development of a program includes usually a number of auxiliary results, which can be used for derivations in the same problem area or for other, related algorithms. Note that in particular every (correct) development forms a transformation rule asserting that the final product fulfils the initial specification. We remark that although the essence of transformational programming lies in the constructive derivation of programs, there is also room for verification, i.e. in some cases it can be more convenient to guess a program/transformation rule and then verify it by deduction.

In order to minimise translation problems, it is an advantage to do the whole development essentially within one language, which therefore has to be sufficiently rich ('wide spectrum') to allow the adequate description of a program/specification during all phases of the whole software development process. However, this is not a strict necessity, and assuming that the language interfaces are clearly defined it is also feasible to use for example a different language in the requirements building phase or for the formulation of the final programs.

Returning to our example, we recall that our aim is to find an efficient implementation of the function *update*. According to our methodology, this will be done by successive application of transformation rules which are presented informally (for a rigorous treatment of the rules, we refer the reader to Ref. 27). In order to obtain an applicative program from the descriptive, non-operational specification, we first generalise the problem by introducing two new parameters s and s'. Intuitively, these will record the progress made in updating, which is formally captured by the assertion in the new function *upd*.

Step 1: generalisation (with assertion).

```
function update (b:board):board;
  upd(ε, ε, b) where
function upd (s, s', t: board || no-gaps (s) ∧ all-gaps(s') ∧
                 same-entries(s ++ s' ++ t, b)):board;
    s ++ s' ++ some u:board || |u| = |t| ∧ same-entries(t, u)
                                         ∧ normalized(u)
```

The parameter t represents that part of the board which has not yet been updated. We first solve the simple case when it is empty.

Step 2: case introduction in *upd*: $isempty(t) \vee \neg isempty(t)$; simplification.

```
function upd (s, s', t:board || no-gaps(s) ∧ all-gaps(s') ∧
                 same-entries(s ++ s' ++ t, b)):board;
  if isempty(t) then s ++ s'
    else s ++ s' ++
       (some u: board || |u| = |t| ∧ same-entries(t, u) ∧
                                      normalized(u))
  endif
```

When t is not empty, we investigate its first element and append it either to s or s', according to the assertion.

Step 3: case introduction in **else**-part: $\neg first(t)$ **is** gap $\vee$ ($first(t)$ **is** gap); simplification.

```
function upd (s, s′, t: board || no-gaps(s) ∧ all-gaps(s′) ∧
            same-entries(s ++ s′ ++ t, b)): board;
  if isempty(t) then s ++ s′
  elsf first(t) is gap
    then s ++ (s′ +< first(t)) ++
      (some u: board || |u| = |rest(t)| ∧
      same-entries(rest(t), u) ∧ normalized(u))
    else (s +< first(t)) ++ s′ ++
      (some u: board || |u| = |rest(t)| ∧
      same-entries(rest(t), u) ∧ normalized (u))
  endif
```

Now the expressions in the second and third alternatives are instantiations of the specification of *upd* as given in step 1. Furthermore, the new parameters comply with the assertion. Therefore folding is possible, whereby we obtain an applicative program. Note that the length of the third parameter of the recursive calls strictly decreases, so termination is ensured.

Step 4: fold (with assertion).

```
function upd (s, s′, t: board || no-gaps(s) ∧ all-gaps(s′) ∧
            same-entries(s ++ s′ ++ t, b)): board;
if isempty(t)          then s ++ s′
elsf first(t) is gap   then upd(s, s′ +< first(t), rest(t))
                       else upd(s +< first(t), s′, rest(t))
endif
```

Notice that the sum of the lengths of the three parameter sequences is constant; so they can be represented by one indexed sequence and two indices denoting the boundaries between the three segments. Furthermore, this suggests an implementation of the sequence involved by means of an array.

Step 5: data type representation (i, k, u) for (s, s', t) with $1 \leqslant i \leqslant k \leqslant |u|+1$ such that s corresponds to $u[1:i-1]$, s' to $u[i:k-1]$, and t to $u[k:|u|]$).

```
type board = ARRAY(elem);
function update (t: board): board;
  upd′(1, 1, t) where
function upd′ (i, k: nat; u: board): board;
  if k > |u|          then u
  elsf u[k] is gap    then upd′(i, k + 1, u)
                      else upd′(i + 1, k + 1, swap(u, i, k))
  endif
```

where *swap*(*u*, *i*, *k*) exchanges the elements on position *i* and *k* in the array *u*.

Basic rules are often combined into more powerful and compact rules. The resulting transformations allow us to capture programming techniques like the simplification of recursion, various methods for storing of values instead of recomputation, and the transition from a tail-recursive definition to an imperative program. The latter can be applied here, leading to our final, imperative program.

Step 6: transition to iteration; sequentialisation.

```
function update = (t: board): board;
  begin
    var vi, vk: nat; var vt: board; var ve: elem;
    vi := 1; vt := t;
    for vk from 1 upto |t| do
      if vt[vk] is gap
        then skip
        else ve := vt[vi]; vt[vi] := vt[vk]; vt[vk] := ve;
          vi := vi + 1
      endif
    enddo
    vt
  end
```

If we compare this program with the one given at the beginning, we realise that they coincide with respect to the overall structure, but differ in the **else**-branch of the conditional. In fact, the program given in the beginning is not correct, as it only works properly when *vk* is different from *vi*. This phenomenon is typical of programming 'in one step'. The essential idea underlying an algorithm is often straightforward, but one fails in getting the details and borderline cases right. The methodology advocated in this paper is a safeguard against making these kinds of errors, the removal of which is known to be enormously time-consuming, if successful at all.

The most inventive step in the above derivation was the choice of generalisation and assertion in the first step. Together with the decision in step 3 to consider the first element of the remaining part of the board, this led in a straightforward way to a linear algorithm which updates the board from top to bottom. By choosing a different generalisation:

```
function update (b: board): board;
  const (board b₁, board b₂) = pupd(b); b₁ ++ b₂;
function pupd (s: board || ¬isempty(s)): (board, board);
  some (t: board, t′: board) || no-gaps(t) ∧ all-gaps(t′) ∧
       same-entries(t, s) ∧ |t| + |t′| = |s|
```

but with an otherwise analogous derivation, one arrives at the following (recursive) 'divide-and-conquer' solution

```
function pupd (s: board || ¬isempty(s)): (board, board);
  if |s| = 1 then if first(s) is gap then (ε, s) else (s, ε) endif
      else  const (s′: board, s″: board) = nonempty-
              split(s);
            const (t₁: board, t₂: board) = pupd(s′);
            const (t₃: board, t₄: board) = pupd(s″);
            (t₁ ++ t₃, t₂ ++ t₄)
  endif
```

This algorithm relies on other operations of our data type, and thus needs another implementation in order to be efficient. Whereas the former solution was well suited for implementation by arrays, this one can be made efficient by using linked lists and pointers. Note furthermore that this algorithm is a good starting point for parallelisation.

4. STATE OF THE ART AND NEW DEVELOPMENTS

At first glance, one might get the impression that not much progress has been made in the area of transformational programming since Burstall and Darlington's seminal work[11] and other results from the late 1970s and early 1980s. Indeed, in many approaches the core transformation steps *fold*, *generalise* and *unfold* still play a crucial rôle – just as induction and generalisation still do in mathematical proofs. Actually, progress has been made in multiple ways.

Theoretical foundations and abstract descriptions of program transformations and their underlying concepts have led to deeper insights into the nature of computing and programming. The ideas of transformational (or calculational) programming have shown to be applicable in various areas besides the optimisation of functional programs. Numerous research groups, many of them liaised with IFIP Working Group 2.1 ('Algorithmic Languages and Calculi'), are working in this field, and research projects at national and international levels have brought about further cooperation and increase of knowledge. Various conferences and journals devoted entirely to formal program development also bear witness to interest and progress in the area.[10, 24, 32, 38, 39] Stimulated by this positive evolution, it is necessary and worthwhile at this time to initiate and stimulate a further transfer of the collected knowledge to practitioners, and to researchers in other, related areas.

4.1 Specification and languages

Most research groups in the area of formal program development nowadays seem to agree that functional languages are easiest for use in formal manipulations. Higher-order functions play an ever-increasing role, often in conjunction with a semantics and calculus based on category theory, which allow us to capture calculational properties in a very general way.[22, 23] Much of this framework can also be retained in a relational setting.[1] Recent research has shown that particular classes of higher-order functions, so-called skeletons, provide useful abstractions of parallel architectures to be used in descriptions and derivations of parallel algorithms.[15, 25]

In general, progress made on the language level will also have an impact on the effort to be spent in requirements engineering, i.e. in obtaining specifications: development of more abstract and more expressive specification constructs widens the scope of formal program development, and thus lessens the effort needed in requirements engineering, although, admittedly, there is still some work to be done in this area.

Abstract data types have become an established subject in theoretical computer science.[41] 'Automatic' implementations of abstract data types, e.g. via term rewriting, provide a valuable prototyping possibility. Libraries of often-used data structures (including various standard implementations) will form a substantial part of the future programmer's workbench. Abstract data types are making their way into industrial applications, in successful formal specification formalisms like VDM and Z,[40] and via their (limited) implementations in Ada and object-oriented languages.[13]

4.2 Methodology

A theoretical basis for program development by transformation has been given in the form of a transformational calculus.[33] Currently the emphasis is on the consolidation, classification, and 'transfer' to practitioners of the available knowledge on program transformations, as summarized in (e.g.) Refs 18 and 27. Even though most results have been obtained in the area of deriving efficient functional and imperative algorithms for sequential architectures and particular languages, it is becoming more obvious that 'transformational programming' is actually a 'meta-approach', which is largely independent of the language or programming paradigm used. The application of the method with various other paradigms, like logic programming and object-oriented programming, has led to some interesting first results. In particular, the application to parallel programming (Refs 15 and 25) seems promising. Always important is the investigation into higher-level steps in transformational developments: tactics and strategies.[18, 36] A complementary approach is the investigation into reusability of transformational developments. Some first case studies have been done by our group.[28, 31] Another aspect that needs further research is the complexity effect of transformations,[42] – even though transformational programming purports to aim at more efficient programs, usually complexity considerations are not formally taken into account during developments.

4.3 Systems

Many small transformation systems have been implemented in order to test the feasibility of the approach.[21, 29] Most of these are of an experimental, academic, nature. Promising results have been achieved with Smith's KIDS system.[35] Current emphasis is also on systems for supporting the specification process.[20] More sophisticated transformation systems can also provide on-line support during the derivation, thus helping to establish not only correctness but also other program qualities like efficiency, modularity or robustness. Ultimately, this should lead to 'expert transformation systems', which perform derivations (semi-)automatically. On a more prosaic level, the computer can be used to provide complete documentation of the derivation process.[7] For the future, transformation systems are envisaged as components of integrated project support environments,[8] along with components like compilers, (syntax-directed) editors and theorem provers.

4.4 Applications

Many of the standard algorithmic problems of computing science have by now been treated in a transformational way, like sorting,[9, 14] parsing[26] and pattern matching.[5, 30, 34] Further examples have been drawn from other traditional example areas, like combinatorics and graph problems. Less experience exists with formal specification and development of medium-sized and large software systems, the development of a transformation system within the CIP project being an exception.[3] It is now recognised that there is a need to apply this method to more examples of everyday programming.

5. CONCLUSIONS

At first glance, it seems that transformational programming has not provided the solutions to the software problem which had been envisaged by early researchers in the area.[2, 12] In particular, science has not yet succeeded in providing the powerful and easy-to-use support systems that are vital to a realistic and large-scale application of the method. To a large extent this is due to the fact that most published work is still too 'academic' and not usable by practitioners as it stands. Another problem is that program transformations constitute a relatively *new* calculus, whose complexity has been likened to the complexity of integral calculus but which is, however, not yet as widely accepted.

On the other hand, research in program transformation has led to the discovery of a large amount of knowledge on the process of programming and the algorithmic structure of programs. A growing number of people have become aware of formal methods for specification and development of programs, and even if they do not apply such methods all the time, they will have gained a deeper insight and broader perspective on the programming activity.

REFERENCES

1. R. C. Backhouse, P. J. de Bruin, G. Malcolm, E. Voermans and J. van der Woude, Relational catamorphisms. In Ref. 39, pp. 287–318.
2. R. Balzer, T. E. Cheatham, Jr and C. Green, Software technology in the 1990s: using a new paradigm. *IEEE Computer*, **16** (11), 39–45 (1983).
3. F. L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner and P. Pepper, *The Munich Project CIP. Volume II: The Transformation System CIP-S*, Lecture Notes in Computer Science 292. Springer-Verlag, Berlin (1987).
4. J. A. Bergstra, J. Heering and P. Klint (eds), *Algebraic Specification*. ACM Press, New York (1989).
5. R. S. Bird, J. Gibbons and G. Jones, Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, **12**, 93–104 (1989).
6. R. S. Bird and P. L. Wadler, *Introduction to Functional Programming*. Prentice-Hall, Hemel Hempstead (1988).
7. E. A. Boiten, M. G. J. van den Brand, N. W. P. van Diepen, C. H. A. Koster, H. A. Partsch and N. Völker, USTOPIA requirements – thoughts on a user-friendly system for transformation of programs in abstracto. *Periodica Polytechnica Series on Electronic Engineering*, **35** (2), 101–123 (1992).
8. A. W. Brown, *A View Mechanism for an Integrated Project Support Environment*. Report 275, University of Newcastle upon Tyne (1988).
9. M. Broy, Program construction by transformations: a family tree of sorting programs. In *Computer Program Synthesis Methodologies*, edited A. W. Biermann and G. Guiho, pp. 1–49. Reidel, Dordrecht (1983).
10. M. Broy and M. Wirsing (eds), *Methods of Programming*. Lecture Notes in Computer Science 544, Springer-Verlag, Berlin (1991).
11. R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *Journal of the ACM*, **24** (1), 44–67 (1977).
12. R. M. Burstall and M. S. Feather, Program development by transformation: an overview. In *Les fondements de la programmation*, edited M. Amirchahy and D. Neel. Proc. Toulouse CREST Course on Programming, IRIA-SEFI, Le Chesnay, France (1978).
13. S. Danforth and C. Tomlinson, Type theories and object-oriented programming. *ACM Computing Surveys*, **20** (1) (1988).
14. J. Darlington, A synthesis of several sort programs. *Acta Informatica*, **11** (1), 1–30 (1978).
15. J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J. Kelly, K. M. Sephton and D. W. Sharp, Structured parallel functional programming. In *Proceedings of the Third International Workshop on the Implementation of Functional Languages on Parallel Architectures*, edited H. Glaser and P. Hartel, pp. 31–51 (1991).
16. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1: Equations and Semantics*. Springer-Verlag, Berlin (1985).
17. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 2: Module Specifications and Constraints*. Springer-Verlag, Berlin (1990).
18. M. S. Feather, A survey and classification of some program transformation approaches and techniques. In Ref. 38, pp. 165–196.
19. A. J. Field and P. G. Harrison, *Functional Programming*. Addison-Wesley, Reading, Mass. (1988).
20. J.-P. Finance and J. Souquières, *A Development Process Meta-model*. Technical Report CRIN-91-R-009, Centre de Recherche en Informatique Nancy (1991).
21. B. Krieg-Brückner, The PROSPECTRA Methodology of Program Development. In *IFIP/IFAC Working Conference on Hardware and Software for Real Time Process Control. Warsaw, Poland*, pp. 1–15. North-Holland, New York (1988).
22. G. R. Malcolm, Data structures and program transformation. *Science of Computer Programming*, **14**, 255–279 (1990).
23. E. Meijer, M. M. Fokkinga and R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming and Computer Architecture*. Lecture Notes in Computer Science 523, pp. 124–144. Springer-Verlag, Berlin (1991).
24. J. L. A. van de Snepscheut (ed.), *Mathematics of Program Construction, Proceedings*. Lecture Notes in Computer Science 375, Springer-Verlag, Berlin (1989).
25. R. Paige (ed.), *Proceedings Workshop on Parallel Algorithm Design and Transformational Programming*, 1992. Kluwer, Deventer (to appear).
26. H. Partsch, Transformational program development in a particular problem domain. *Science of Computer Programming*, **7** (2), 99–241 (1986).
27. H. Partsch, *Specification and Transformation of Programs – a Formal Approach to Software Development*. Springer-Verlag, Berlin (1990).
28. H. A. Partsch and E. A. Boiten, A note on similarity of specifications and reusability of transformational developments. In Ref. 39, pp. 71–89.
29. H. A. Partsch and R. Steinbrüggen, Program transformation systems. *ACM Computing Surveys*, **15** (3), 199–236 (1983).
30. H. A. Partsch and F. A. Stomp, A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, **2**, 109–122 (1990).
31. H. A. Partsch and N. Völker, Another case study on reusability of transformational developments – pattern matching according to Knuth, Morris, and Pratt. In Ref. 10, pp. 36–48.
32. P. Pepper (ed.), *Program Transformation and Programming Environments*. NATO ASI Series, Series F: Computer and System Sciences, Vol. 8. Springer-Verlag, Berlin (1983).
33. P. Pepper, A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, **9** (3), 221–262 (1987).
34. P. Pepper, Literate program derivation: a case study. In Ref. 10, pp. 101–124.
35. D. R. Smith, KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, **16** (9), 1024–1043 (1990).
36. D. R. Smith and M. R. Lowry, Algorithm theories and design tactics. *Science of Computer Programming*, **14**, 305–321 (1990).

37. I. Sommerville, *Software Engineering*. Addison-Wesley, Reading, Mass. (1992).
38. L. G. L. T. Meertens (ed.), *Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation*. North-Holland, Amsterdam (1987).
39. B. Möller (ed.), *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*. North-Holland, Amsterdam (1991).
40. S. Prehn and W. J. Toetenel (eds), *VDM '91 Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands*. Lecture Notes in Computer Science 551 and 552, Springer-Verlag, Berlin (1991).
41. M. Wirsing, Algebraic specification. In *The Handbook of Theoretical Computer Science*, edited J. van Leeuwen, pp. 675–788. Elsevier, Amsterdam (1990).
42. W. Zimmermann, *Automatische Komplexitätsanalyse funktionaler Programme*. Informatik-Fachberichte 261, Springer-Verlag, Berlin (1990).

Book Review

F. D. Rolland, *Programming with VDM*, Macmillan Press, £13.99. ISBN 0-333-56520-7

The author describes this book as 'an introduction to VDM and how it can be used by the programmer'. It begins with a broad overview of formal methods in general and VDM in particular, including a section promoting the use of formal methods as a tool for improving the quality and reliability of software. This chapter also points out the parallel between formal specification and object-oriented design.

Chapters 2 to 7 introduce the various basic elements of the VDM language. The first of these presents a much simplified view of basic logic and simple set theory, and this is followed by chapters dealing with functions, operations and states, composite objects, sequences and maps. Each of these last five chapters includes a simple VDM specification illustrating the use of the particular VDM construct it introduces, accompanied by an 'implementation' of it as a program in Modula-2. This pattern is repeated in the final chapter, which presents a rather more substantial case study making use of all the concepts dealt with in the earlier chapters.

The author's enthusiasm for formal methods is evident throughout, and the text of the book is generally easy to read. Unfortunately the same is not always true of the mathematical formulae or of the VDM. In particular I find the curious mixture of mathematical notation, in which some special characters appear verbatim (e.g. $\subseteq$, $\in$, $\wedge$) whereas others appear as some sort of Ascii approximation (e.g. $\neq$, $\Rightarrow$ and $\geqslant$ appear as #, = = > and > = respectively), annoying. In addition there are rather too many typographical errors.

More serious worries attach to the semantic (in a mathematical sense) content of the material presented. The first problem is that the treatment throughout could easily be taken to imply that classical Boolean logic and not the logic of partial functions (LPF) is the true foundation of VDM – there is virtually no mention of the notion of undefinedness which is central to LPF.

Another problem is that the example specifications presented seem to fail entirely to capture the idea that a formal specification gives a way of defining the functionality of a piece of software abstractly without having to worry about details of interface, hardware, etc. – too many of the examples get hung up on details of how the results of operations are presented to the end user's computer screen, whereas I would have preferred such details to have been left to the implementation. In a way this is perhaps understandable in that the author is clearly trying to present pairs of specification/code which are as close to each other as possible, but I feel that this treatment obscures too much of the power of formal specification. Indeed, the impression is that the specifications have been reverse-engineered from the code rather than that the specifications have been written so as to encapsulate the essential functionality of the system and that the code has then followed.

The most serious problem by far, however, is that the author does not seem to fully understand the semantics of the specifications he writes, as several of the examples are faulty. For instance, one defines a state which can only take one value, namely empty, with the result that all the operations defined on this state are actually unimplementable; another has an under-defined operation.

The stated aim of the book is 'to enable readers not only to write and interpret simple VDM specifications, but also to use a simple set of guidelines for transforming these specifications into a high-level language similar to Modula-2'. On the assumption that the Modula-2 code given as implementing the faulty specifications actually implements what the specification was meant to say as opposed to what it does say (my knowledge of Modula-2 is minimal), then it is perhaps as well, if disappointing, that I do not believe these guidelines are apparent. And sadly there are too many problems with the specifications for it to be recommended for serious students of VDM, whether programmers or otherwise.

R. Moore
Manchester

Special issue on Spatial Data – Call for Papers

Later in 1993 *The Computer Journal* will be publishing a special issue on **Spatial Data**. This is a particularly important subject at the present time and the issue is seen as being a major contribution to the scientific literature. For this reason a formal Call for Papers is being issued. Papers are invited related to techniques for the processing, storage or retrieval of spatial data or related applications involving spatial data. Novel technical contributions, surveys and tutorial papers all will be considered. They must be written for a general audience of computer scientists, not just for specialists. Please send all contributions to the Editor-in Chief at the address given on the front inside cover of this Journal. **Deadline**: 15 March 1993.