Flexible Algorithms - Introducing Parallelism to Beginners

R. B. Yehezkael, M. Goldstein, D. Dayan, S. Mizrahi
Flexible Computation Research Laboratory
Jerusalem College of Technology

e-mail: flexcomp@jct.ac.il Internet: http://flexcomp.jct.ac.il

June 2017 – סיון תשע״ז

This work is based on two papers [1, 2] on this topic appearing in IEEE Distributed Systems Online, and the course notes [3]. It is targeted at teachers who wish to introduce parallelism to computer science students at the very beginning of their studies and is structured as follows: Part 1 gives an overview of the course notes and experiences of the first author, Part 2 contains fragments from the course notes which should clarify the overview in Part 1, and Part 3 contains material to give a taste of things to come.

PART 1 - AN OVERVIEW OF THE COURSE NOTES AND EXPERIENCES

First steps in computer science usually deal with sequential algorithms and programs. Unfortunately, this gets students accustomed to sequential thinking, making it hard for them to understand and effectively use parallelism later. On the other hand, because parallel programming is so difficult, introducing it at an early stage is impractical. So, we developed course notes on flexible algorithms [3] for beginners. Flexible algorithms have a simple notation and are multifaceted. You can execute them sequentially or in parallel, producing results independent of the execution order.
We aim to teach students to
● read and execute flexible algorithms and understand their behaviors,
● acquire an early awareness of parallelism,
● convert flexible algorithms to sequential algorithms,
● use computational induction and thereby better understand flexible algorithms,
● make changes to flexible algorithms, and
● write simple flexible algorithms.
To arouse the students' curiosity and broaden their outlook, we show how flexible algorithms can be converted to hardware block diagrams and briefly show how they can be used for overall system descriptions.

The course notes

The course notes deal with algorithms and *not* with programming. The students encountered parallel execution in the first lecture and this was presented before sequential execution. Exercises were solved by students without the use of a computer. In view of the elementary nature of the material, efficiency of parallel execution was not discussed. The course notes were taught in parallel to a beginner's course on programming where students were learning about basic concepts, such as assignment, conditional statements, etc.
The first version of the course notes was too complex notationally. So, students and teachers had problems. The second version was notationally simpler and both students and teachers were satisfied. A total of three hours per week for a semester are needed to teach this material, part lectures and part exercises.

Educational approach
Should the first steps we teach deal with sequential or parallel algorithms and programs? Should they deal with software or hardware? Should we discuss small-scale matters (such as algorithms and logic design) or large-scale ones (such as systems)? Because the course notes are intended for complete beginners, our approach is small scale, but we differ by starting with flexible algorithms.

The form of Flexible algorithms
The core algorithmic language of flexible algorithms is based on functions with IN parameters and OUT parameters (but no IN/OUT parameters), conditional statements and once only assignment statements. These restrictions make the statements in the flexible algorithm independent of each other and enable execution in a variety of orders, parallel and sequential, giving unique results. Blocks and loops (including nested forms) aren't part of the core language, but are abbreviations for certain compound forms in the core language. The *parallelism* in our language is *implicit* and *optional*. On the surface, our language has a sequential like imperative style, and in a way parallelism is hidden.

Execution methods
In the course notes three execution methods are described:
● parallel,
● sequential with immediate calls, and
● sequential with delayed calls.
With all methods, the execution is presented in the same form: that is, as sets of statements with values of results. The execution order is implicit; there are no statements for forcing parallel or sequential execution.

Hardware block diagrams
We present an algorithm for adding serially bit by bit (or digit by digit) and an example of its execution. We then transform this algorithm into a hardware block diagram of the type found in books on logic and digital systems. The techniques for executing and generating a hardware block diagram are similar, and both techniques process sets of statements in similar ways.

Different styles for passing parameters
Here's a fragment of a flexible algorithm written as a function:

```
function v' •= reverse(v, low, high);
    {
    ...
    v' •= reverse(v, low + 1, high –1);
    ...
    } // end reverse
```

This fragment includes a call or activation of itself, which we wrote functionally. We can equivalently write such a call in an abbreviated assignment style, showing only the parameters changed:

```
reverse(low •= low + 1; high •= high –1);
```

The statements low •= low + 1; high •= high – 1 don't change the values of low and high. Here, low on the right side of the assignment, denotes the current variable low, and

low on the left side denotes the new variable low that will be used by the new activation of the function reverse. The same is true for the variable high. So, each call or activation of the function reverse has separate variables.
The students quickly caught on to how several variables can have the same name. This is no more difficult than dealing with a number such as 999, where the leftmost 9 denotes the value 900, the middle 9 denotes the value 90, and the rightmost 9 denotes the value nine. That is, the value of the digit 9 depends on its position or context. Similarly, a variable on the right hand side of •= is an existing variable, and one on the left hand side of •= is a new variable with the same name.

Conversion to a sequential algorithm
There are two reasons for presenting material as to how to convert a flexible algorithm to a sequential algorithm.

a. Starting with a flexible algorithm and converting it to a sequential algorithm can produce a sequential algorithm where the possibility of some parallelism remains. This parallelism may be exploited by multi-scalar processors and cores to automatically execute parts of this sequential algorithm in parallel. More on this in Part 2.
b. So as to prevent the teaching of this material from becoming divorced from the material of a beginners' course on sequential programming which was taught in parallel to this material.

The abbreviated assignment style is important in making the conversion, because it provides statements that look on the surface like assignment statements. They're good candidates for being rewritten as actual assignment statements that update a variable's value.

Computational induction
Computational induction allows you to assume that internal calls work according to specification, and thus allows you to ignore the details of all these sub-computations. The rule that variables are either IN or OUT, and the use of once only assignment makes function values independent of execution order. Putting the two together, simplifies the analysis of flexible algorithms. Computational induction does not have an explicit basis like zero or one which is used for simple induction. This is why it does not prove termination of execution. An analogy: in mathematics a proof is sometimes divided into two parts, an existence proof and a uniqueness proof. Computational induction is like a uniqueness proof, there is no proof that the program will terminate and produce a well-defined result. Students were taught how to use computational induction without justifying why it works. For further information about computational induction see [6, 7].

Labeled and unlabeled blocks and local variables
Such blocks are convenient abbreviations for internal functions that let us easily introduce local variables and loops. This makes writing algorithms simpler, but we delay presenting this material until the student has mastered the basic form of flexible algorithms and proofs by computational induction.
We also present a forall loop and various equivalent forms, including one that enables (but doesn't force) parallel execution of all iterations.

For example, forall i •= m, n; {statements} is equivalent to this set of unlabeled blocks:

```
{let i •= m; statements}
{let i •= m + 1; statements}
...
...
{let i •= n – 1; statements}
{let i •= n; statements}
```

Each iteration of the forall loop has a separate local variable i, and this enables parallel execution of all the iterations.

Reactions to our approach

Some colleagues thought that we should teach such material after students mastered sequential algorithms and programs. Perhaps they were right concerning the first version of the material, but not its second version. Indeed, after the college instituted a common first semester for all engineering departments, it decided that this material wasn't of sufficient interest to all departments and moved it to the second semester after teaching sequential algorithms and programs. Our material was then found to be too easy, indicating that our material is at the right level for beginners. Another colleague thought the opposite—that such material should be taught before sequential algorithms and programs.

As mentioned earlier, students found the first version difficult but enjoyed the second version. Here are some students' comments:

- "This course is better than the companion course on sequential algorithms and programs because there's the possibility of using parallelism."
- "In time, everyone will know languages such as C++ and Java. The knowledge of this kind of material regarding parallelism is an advantage in finding work."
- "It's amazing that you can derive a hardware block diagram of a serial adder from a flexible algorithm for addition."
- "This course is superfluous."

The course notes in the PDC curriculum

The course notes [3] are most relevant for the first CS1 course in the PDC curriculum. The following table summarizes its educational impact.

General topic **Section No. and Title**	**Bloom's** classification	**Learning Outcome**
Algorithmic Paradigms		
1. Introduction	C,A	Motivation of the students.
2. Flexible algorithms in a functional style	C	Understand the form of flexible algorithms.
3. Different styles for passing parameters	C,A	Understand the two primary styles for parameter passing.
5. Converting flexible algorithms into a sequential algorithm	C,A	Flexible algorithms have equivalent sequential algorithms.

8. Convenient abbreviations for writing functions	C,A	Unlabeled and labeled blocks allow the introduction of local variables and recursion with an iterative style. Parallel loops are introduced too.
Algorithmic Problems		
7. Various flexible algorithms	A	Reduction, Divide and Conquer
Semantics and correctness issues		
6. The correctness of (flexible) algorithms	C,A	Comprehension of the use of computational induction for proving partial correctness of algorithms.
Cross Cutting issues		
4. Converting flexible algorithms to hardware block diagrams	C	Understand that certain flexible algorithms may be converted to hardware block diagrams.
9. Overall description of systems using flexible algorithms	C	Understand that using flexible algorithms to describe a simple system leaves options open for design.

Concluding remarks

To help students exploit parallelism effectively, it is important that they learn about it in a simplified form at the beginning of their studies. After learning this material, students could easily execute flexible algorithms in various orders (including parallel execution), and make small changes to these algorithms. They had little difficulty converting a flexible algorithm to a sequential one. Our notation's imperative style with once only assignment, should made it easier for students to make the transition to imperative programming, including imperative parallel programming. Writing flexible algorithms from scratch was harder for students, as was computational induction.

Flexible algorithms are important in design. They may be converted to hardware block diagrams, see the course notes [3]. This facet of flexible algorithms is important given the availability of programmable hardware. It allows the use of a flexible algorithm as a design which may be later implemented as a program or in hardware. It leaves design options open. We also showed, how using a flexible algorithm to describe a simple system left design options open. These topics were presented to arouse the student's curiosity and broaden his or her outlook, which is important for first steps in Computer Science.

PART 2 - FRAGMENTS FROM THE COURSE NOTES [3]

Execution of Flexible Algorithms

Execution of flexible algorithms by three methods, are given in the course notes:

- parallel execution,
- sequential execution with immediate execution of function calls or activations, and
- sequential execution with delayed execution of function calls or activations.

To save space, parallel execution only is presented here:

Reversing five values

For educational reasons, we presented a specific solution for five values before giving a function for reversing part of a vector. We also presented three execution methods for reversing a five-element vector using this general solution. See the course notes for details.

The following is a flexible algorithm for reversing five values.

```
function a', b', c', d', e' •= rev5(a, b, c, d, e);
// SPECIFICATION:
// IN - a, b, c, d, e are any values.
// OUT - values a', b', c', d', e' like a, b, c, d, e but in reverse order
{ a'•=e;
  b', c', d'•=rev3(b, c, d);
  e'•=a;
} // end rev5
function a', b', c' •= rev3(a, b, c);
// SPECIFICATION:
// IN - values a, b, c
// OUT - values a', b', c' like a, b, c but in reverse order
{ a'•=c;
  b' •=rev1(b);
  c'•=a;
} // end rev3
function a' •= rev1(a);
// SPECIFICATION:
// IN - value a
// OUT - value a' like a but in reverse order - that is a' is a
{ a'•=a;
} // end rev1
```

Suppose we wish to execute the statement a', b', c', d', e' •= rev5 (1, 2, 3, 4, 5). We begin by writing this as a set of one element, namely { a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }. The steps of the parallel execution are given below.

set of statements	a', b', c', d', e'
{ a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }	_, _, _, _, _
{ a' •= 5 ; b', c', d' •= rev3(2, 3, 4) ; e' •= 1 }	_, _, _, _, _
{ b' •= 4, c' •= rev1(3) ; d' •= 2 }	5, _, _, _, 1
{ c' •= 3 }	5, 4, _, 2, 1
{ }	5, 4, 3, 2, 1

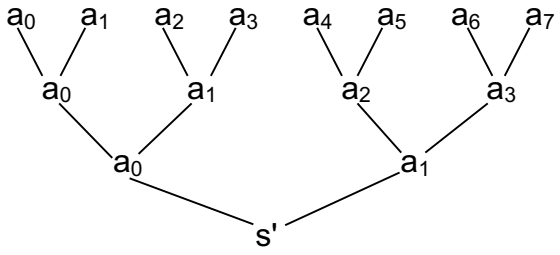
For educational reasons, we presented a specific solution for five values before giving a function for reversing part of a vector. We also presented three execution methods for reversing a five-element vector using this general solution. See the course notes for details.

Summing eight numbers

The statement $s' \bullet= a_0 + a_1 + \ldots\ a_7$ is equivalent to $s' \bullet= ((\ldots(a_0 + a_1) + a_2) + a_3) + \ldots a_7)$ when fully bracketed. When bracketed in this way, sequential evaluation is forced. Here is a different bracketing of the expression which enables parallel evaluation.

$s' \bullet= (((a_0 + a_1) + (a_2 + a_3)) + ((a_4 + a_5) + (a_6 + a_7)))$

Intuitively, here is how the calculation is performed.



In the above, it is useful to view each occurrence of a_i as a separate variable. This view simplifies writing this calculation as a flexible algorithm, in which the computation of sub-expressions may be performed in parallel. So for example, there are three variables named a_0 , etc. Here now are three functions add8, add4, add2 for carrying out the computation in this way.

function $s' \bullet= add8\ (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$
// SPECIFICATION: $s' = a_0 + a_1 + \ldots\ a_7$
{ add4 ($a_0 \bullet= a_0+a_1$; $a_1 \bullet= a_2+a_3$; $a_2 \bullet= a_4+a_5$; $a_3 \bullet= a_6+a_7$);} // end add8
// The previous call is equivalent to $s' \bullet= add4\ (a_0+a_1, a_2+a_3, a_4+a_5, a_6+a_7)$

function $s' \bullet= add4(a_0,a_1,a_2,a_3)$;
// SPECIFICATION: $s' = a_0 + a_1 + a_2 + a_3$
{add2($a_0 \bullet= a_0+a_1$; $a_1 \bullet= a_2+a_3$);} // end add4
// The previous call is equivalent to $s' \bullet= add2\ (a_0+a_1, a_2+a_3)$

function $s' \bullet= add2(a_0, a_1)$;
// SPECIFICATION: $s' = a_0 + a_1$
{$s' \bullet= a_0+a_1$}; // end add2

Here is the parallel execution of $s' \bullet= add8(1,2,3,4,-1,-2,-3,-4)$.

set of statements	*s'*
{s' •= add8(1,2,3,4,-1,-2,-3,-4)}	_
{s' •= add4(3,7,-3,-7)}	_
{s' •= add2(10,-10)}	_
{ }	0

Summing the elements of a vector of size n where n is a power of 2
Here is how the previous specific example can be generalized to handle a vector of n elements, where n is a power of 2.

```
function s'•=addn(v,n)
// SPECIFICATION:v is a vector and n is its size which must be a power of 2.
//                    The function computes s' = v0 + v1 ... + vn-1
{
if    (n=1)
  {s' •= v0}
else addn (
            n•=n/2;
            v0•=v0+v1;
            v1•=v2+v3
            .
            .
            .
            vn/2-1 •= vn-2 + vn-1
           );
} // end addn
```

Here is the parallel execution of s' •= addn((1,2,3,4,-1,-2,-3,-4), 8).

set of statements	*s'*
{s' •= addn((1,2,3,4,-1,-2,-3,-4), 8)}	_
{s' •= addn((3,7,-3,-7), 4)}	_
{s' •= addn((10,-10), 2)}	_
{s' •= addn((0), 1)}	_
{ }	0

Another simple exercise is to write a flexible algorithm to find a vector's minimum value using the structure of the function addn. The students need only to replace the operator + with a function min for determining the minimum of two values and then write the definition of the function min.

Merge sort
We use the structure of the function addn to develop the bottom-up merge-sort algorithm. This gives students a gentle introduction to a subtle sorting technique that enables parallel sorting of a vector. This technique sorts a vector by repeated merging. For simplicity we assume that the length of the vector is a power of 2. For example suppose we wish to sort a vector of 8 elements. We view this as 8 single elements, and of course each single element is sorted.

(8, 1 , 7, 2 , 6, 3 , 5, 4)

We merge pairs of single elements to obtain:

(1,8 , 2,7 , 3,6 , 4,5)

Now we have four sorted pairs, so we merge two and two pairs to obtain:

(1,2,7,8 , 3,4,5,6)

Now we have two sorted runs of four elements, so we merge them to obtain:

(1,2,3,4,5,6,7,8)

Now we have a sorted vector.

Note that at each stage the size of the sorted run in the vector is doubled with respect to the previous stage. Also the number of sorted runs in the vector is halved with respect to the previous stage. Here is what happens to these values.

Number of sorted runs	*Size of sorted run*
8	*1*
4	*2*
2	*4*
1	*8*

Exercise
Write a function m2 with specification as follows.

```
function v'•=m2 (v, size, place);
// SPECIFICATION:
// v is a vector having two ranges which are sorted in non-decreasing order.
// These ranges are of length "size" and at position "place" onwards in v.
// These two ranges are merged into a single range of length "2×size"
// and are put at position "place" onwards in v'.
```

Let us now use the function m2 to define the function mergesort. The structure of mergesort is similar to the function addn we defined earlier.

```
function v'•=mergesort(v)
// SPECIFICATION:
// v, v' are vectors having the same length which must be a power of 2.
// v' will be like v but sorted in non-decreasing order.
{loop(size•=1);}    // Short for v'•=loop(v, 1)

function v'•=loop(v, size)
// SPECIFICATION:
// v, v' are vectors having the same length which must be a power of 2.
// The vector v consists of consecutive sorted ranges of length "size"
// which must be a power of 2.
// v' will be like v but but sorted.
{
 if (size=length of v)
   {v'•=v}
  else
     loop (
            size•=size×2;
            v•=m2( v, size, 0 );
            v•=m2( v, size, 2×size );
            v•=m2( v, size, 4×size );
            v•=m2( v, size, 6×size );
            . . .
            v•=m2( v, size, length of v – (2×size) );
          );
 }
```

Converting flexible algorithms to hardware block diagrams

The course notes contain both a textual (hard to understand) and diagrammatic (easy to understand) development of a hardware block diagram for a serial adder. Here we only give the diagrammatic development.

Adding two numbers and a carry

We assume that there is a function a3b for adding three digits producing two results, a sum digit and a carry digit. For example, executing c' , s' •= a3b(9,3,1) would make s'•=3 and c'•=1. (Here s' is the sum and c' is the carry, each being a single digit.) Here is a specification of the function a3b.

```
function c', s' •= a3b(u, v, c)
// SPECIFICATION:
// IN - u, v, c, are the digits to be added.
// OUT - c' is the one digit carry and s' is the one digit sum.
```

Here is a function add for adding two numbers consisting of several digits and an existing carry (a single digit). The function add gives two results: a sum consisting of several digits and a carry consisting of one digit. It is assumed that both numbers being added and the sum produced consist of n+1 digits, where n denotes the position of the least significant digit. The most significant digit has position zero.

```
function c', s' •= add(u,v,c,n);

// SPECIFICATION:
// IN - "n" denotes the position of the least significant digits of u,v, to be added,
// where n≥0. The most significant digit has position zero.
// Digit "c" is also added at the position of the least significant digits of u, v.
// OUT - the carry of the addition is produced in c' and the sum itself in s'.
 {
  if (n>0)
  add( c, s'n•= a3b(un, vn, c); n•=n-1);
  else c', s'0•=a3b(u0, v0, c);
 } // add
```

For example executing c',s'•=add(123,987,6,2) would make s'•=116 and c'•=1 .

Notes

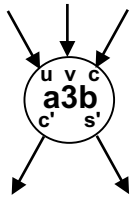
1) In the above function, the occurences of c on opposite sides of the •= denote different variables and similarly for n.
2) Though we have written this function based on decimal digits, the same definition would be used if we used binary digits (or bits).

Here is an example of parallel execution of c',s' •= add(123, 987, 6, 2).

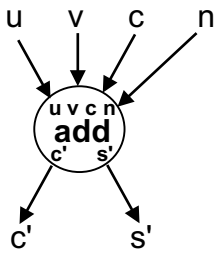
set of statements	c'	s'
u v c n		
{ c',s' •= add(123, 987, 6, 2) }	_,	_, _, _
{ c',s' •= add(123, 987, 1, 1) }	_,	_, _, 6
{ c',s' •= add(123, 987, 1, 0) }	_,	_, 1, 6
{ c', s'_0•=a3b(1, 9, 1) }	_,	_, 1, 6
{ }	1,	1, 1, 6

We now develop from the flexible algorithm a hardware block diagram of a full adder. This is done diagrammatically.

Let us represent a3b diagrammatically by:



A function call c', s'•=add(u, v, c, n) can be represented by:



This diagram is equivalent to one of the following depending whether or not n>0.

Diagram for n>0

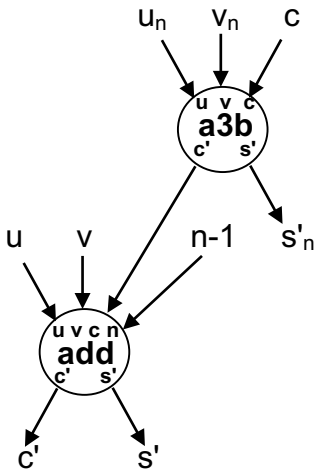
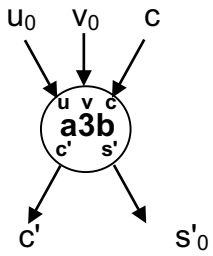
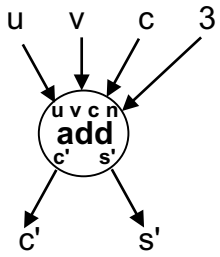


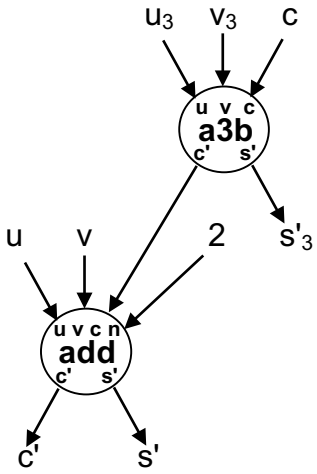
Diagram for n=0



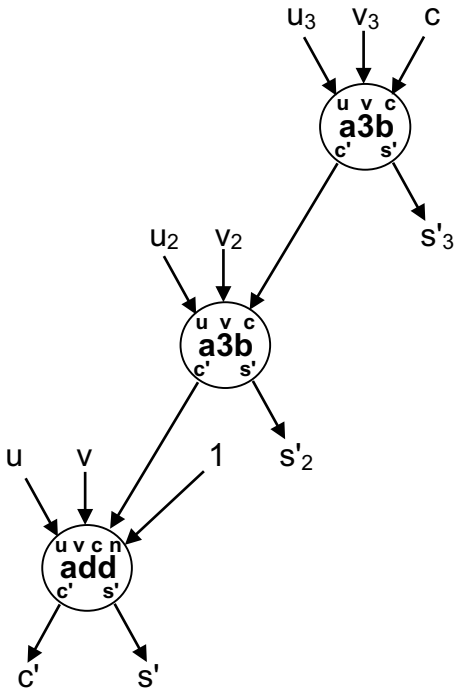
The call for a 4 digit adder is c', s'•=add(u, v, c, 3) and is represented by:



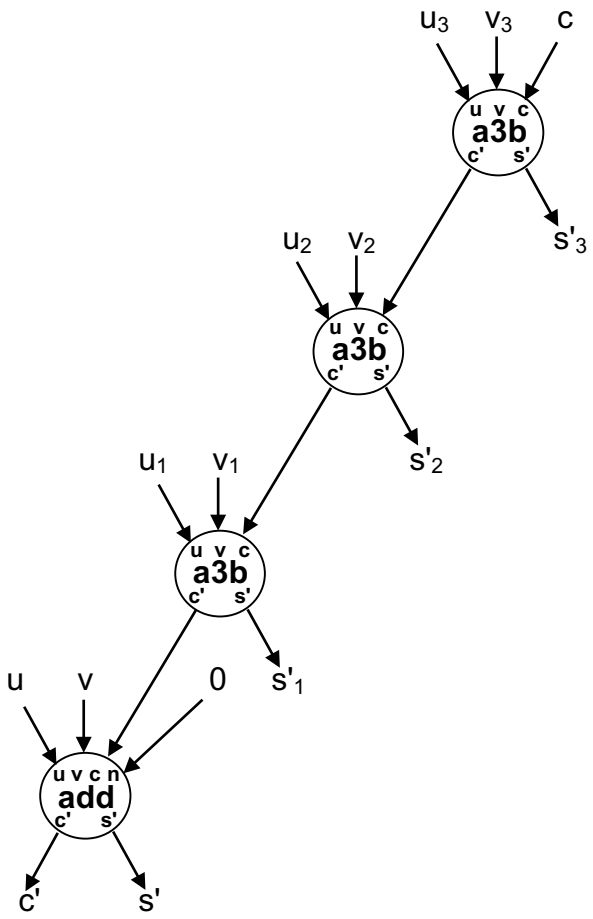
Equivalent to:



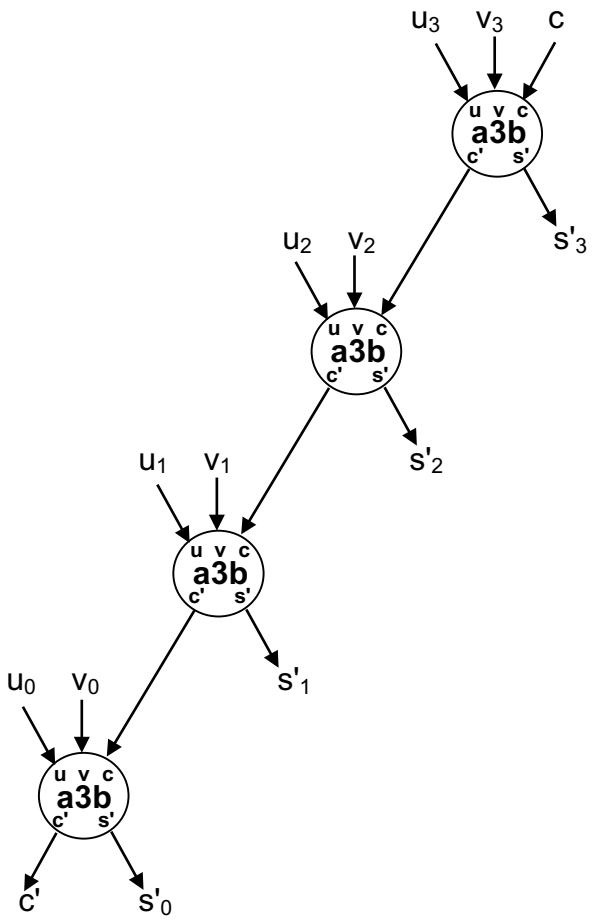
Equivalent to:



Equivalent to:



Finally giving



Here we have obtained diagrammatically, a block diagram for a serial adder.

Converting reverse to a sequential algorithm

The flexible algorithm should first be written in abbreviated assignment style, where in the function calls we only write the values changed. So here is the flexible algorithm reverse written in abbreviated assignment style.

```
function v' •= reverse(v, low, high);

// SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.

{
if (low<high)
   { v'high •= vlow;
     reverse (low•=low+1, high•=high-1);
     // Short for v' •= reverse (v, low+1, high-1).
     v'low •= vhigh;}
else if (low=high)
         {v'high •= vhigh;};
} // end reverse
```

Note
In the above, the assignment low•=low+1, does not change the value of the variable low. This is because low on the right hand side denotes the current variable low, and low on the left hand side denotes the new variable low which will be used by the new activation of the function reverse. (Similarly for high.)

Here is how this flexible algorithm may be converted into a sequential algorithm.

Stage 1
1) Delete the function header function v' •= reverse(v, low, high) and in its place write a label reverse:.
2) Delete the tags from the output variables.
3) In place of reactivating the function by a function call, use a goto statement to the label reverse.
4) Replace •= by :=. Note that •= denotes assignment used in flexible algorithms and does not allow the value of a variable to be changed. However := denotes assignment used in sequential algorithms, and allows many assignments to the same variable and the value of a variable to be changed.

After performing these actions, the sequential algorithm, which is not yet correct, becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// (Additional temporary variables will be used later.)

reverse: { // Need temporary variable(s).
            if low<high
              {
               v_high:=v_low;
              low:=low+1; high:=high-1;
              goto reverse;
               v_low:=v_high    // Statement unreachable.
              }
            else    // Unnecessary and inefficient.
              if low=high
                  v_high:=v_high
          }
```

Stage 2
1) Change the order of the statements suitably.
2) Use temporary variables as needed.
3) Delete unnecessary parts of the algorithm.

After performing these actions, the sequential algorithm becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// Two temporary variables new_vlow, new_vhigh are used below
// to enable enable the swap to be performed correctly.

reverse: { if low<high
           {
             new_vlow:=v_high;   new_vhigh:=v_low;
             v_low:=new_vlow;   v_high:=new_vhigh;
            low:=low+1;   high:=high-1;
            goto reverse;
           }
         }
```

Stage 3

Where possible, use a while loop in place of goto.

After doing this, the sequential algorithm becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// Two temporary variables new_vlow, new_vhigh are used below
// to enable enable the swap to be performed correctly.
while low<high
{
  new_vlow:=v_high;   new_vhigh:=v_low;
  v_low:=new_vlow;   v_high:=new_vhigh;
  low:=low+1;   high:=high-1;
}
```

Notes

1) The above may be executed sequentially, statement after statement. However, the statements on the same line could be automatically executed in parallel by a multi-scalar processor or core.
2) In general, suppose that we write a a flexible algorithm and convert it to a sequential algorithm as above. This will give more opportunities for multi-scalar processors or cores to automatically use parallelism on the derived sequential algorithm when compared to writing a sequential algorithm from the very start. This is because the new variables introduced as in the examples above, give more opportunities for parallel execution.
3) The assignment low:=low+1, changes the value of the variable low, it is increased by one. Here low on the both sides denote the same variable. (Similarly for high, etc.)

Computational induction
This is a proof technique for proving the partial correctness of an algorithm, that is, if the execution halts then the results obtained are in agreement with the specifications. A specification is a precise definition of what the algorithm receives (IN values) and the results it gives (OUT values).

The structure of computational induction
1) Check that every result given explicitly agrees with the specifications.
2) Assume that every internal function call works according to specification and check that the final results the function gives agree with the specifications.
Conclusion: If the execution halts, the results obtained agree with the specifications; that is, the algorithm is partially correct.

Showing that no variable is assigned more than once
In the course notes, we used computational induction to show that a function is partially correct with respect to its specification. Here we shall use computational induction to show that no variable or component of a vector etc. is assigned more than once. In effect, this means we are checking that there are no write/write conflicts, which is important in parallel computing.

Consider this definition of the function reverse.

```
function v' •= reverse(v, low, high);

// SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.

{
if (low<high)
   {v'high •= vlow;
    v' •= reverse (v, low+1, high-1);
    v'low •= vhigh;
   }
else if (low=high)
    {
          v'high •= vhigh;
        }
} // end reverse
```

In the course notes, we used computational induction to show that this function is partially correct with respect to its specification above. Here we shall use computational induction to show that this function is partially correct with respect to the following stripped-down specification, in which we do not describe the result produced but only which elements of v' (the OUT parameter) receive values.

```
// STRIPPED-DOWN SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - No element of v' is assigned more than once.
// If low≤high, then within the range "low" to "high", v' receives values.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.
```

This will show that if the execution halts, no element of v' is assigned more than once; i.e. no write/write conflicts.

Initially the vector v' holds no values.
_ , ··· , _ , _ , ··· , _ , ··· , _

Three cases need to be considered.
(a) low<high. (b) low=high. c) low>high.

a) low<high.
Here we need to execute:

$v'_{high} \bullet = v_{low}$; Transfer value to right.	v'•=reverse(v, low+1 , high-1); This is an internal call and by computational induction we may assume that it works to the stripped-down specification. So v' will receive values in the range low+1 to high-1. Also no element of v' in this range will be assigned more than once.	$v'_{low} \bullet = v_{high}$; Transfer value to left.

v' now is: ..... s_v, s_v, ... s_v, s_v .....
Here s_v denotes some value.
Clearly the entire range from low to high receives values. Also since low < high, the assignments in the left and right columns above do not cause assignments to an element of v' element more than once.
Thus this final value is in agreement with the stripped-down specification of v'•=reverse(v,low,high).

b) low=high.
Here v_{high} is transferred to v'_{high} and v' becomes:
_ , ··· s_v , _ , _ , _
Clearly no element of v' in this range will be assigned more than once.
Thus this final value is also in agreement with the specification of v'•=reverse(v,low,high).

c) low>high.
Here no values are transferred to v' and no element of v' in this range is assigned more than once. This is in agreement with the stripped-down specification of v'•=reverse(v,low,high).

So in all cases the final value of v' is in agreement with the stripped-down specification of v'•=reverse(v,low,high), and reverse is partially correct. This means that if the execution halts, no element of v' is assigned more than once; i.e. no write/write conflicts. Q.E.D.

Labeled "let blocks"

Consider the following functions.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
// SPECIFICATION: Like squares, but puts values only at places i onwards.
{ if (i< length of v)
    { loop(i•=i+1); v'i•=vi×vi; };
};   // end loop
```

It is shorter and more convenient to write the previous function in the following form.

```
function v' •= new_squares (v);
{ let i•=0;  // Initial value of local variable i.
  loop: // This is the label.
  if (i< length of v)
     {
       loop(i•=i+1);     // This activates or calls "loop" again.
       v'i•=vi×vi;
     };
};   // new_squares
```

Note
The use of a label allows us to activate or call the "let block" again from within the "let block". It is a convenient abbreviation for introducing (hidden) internal functions and auxiliary variables.

A modified version of binary search

This version of binary search works with any vector. The two internal calls to new_bsearch may be executed in parallel.

```
function place'•= new_bsearch(v, low, high, value)
// SPECIFICATION:
// v is a vector.
// If "value" is to be found in v in the range "low" to "high", then
// place' will receive the position of the first occurrence of "value" in this range.
// If "value" is not to be found in this range or if low>high,
// then place' is given the value -1.
{
  if ( low>high )
     { place'•= -1; }    // code for not found
  else
     {  let middle•=(low + high)/2;
        if  ( value=vmiddle )
           { place'•=middle }
         else
           {  let place1 •= new_bsearch (v,low, middle-1,value); // search in left half
              let place2 •= new_bsearch (v, middle+1,high,value );    // search in right half
              if place1 ≠ -1
                 { place'•=place1; }    // search in left half found something.
              else
                 { place'•=place2; }    // use value from search in right half.
           }
     }
  }
```

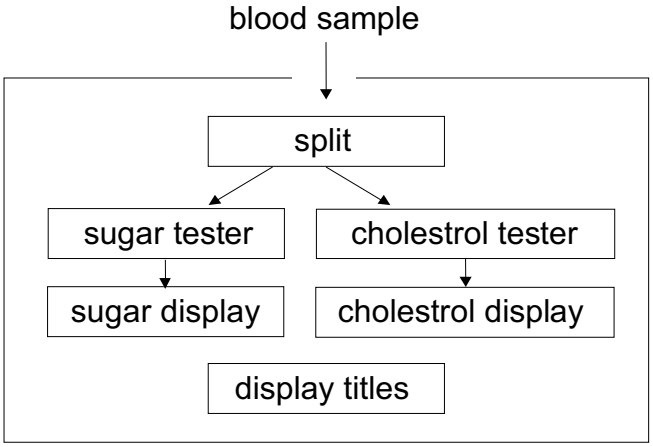
Exercises

1) In the function new_bsearch above, when value is found in the vector v, place' will hold the position of the first occurrence of value in the vector v. Modify the function so that place' will hold the position of the last occurrence of value in the vector v.

2) Rewrite the function new_bsearch above using two auxiliary functions instead of the two "let blocks".

Overall description of systems using flexible algorithms
A system is a collection of components which work together. Some of these components may work independently of each other, in parallel. Other components may need to be synchronized for them to work correctly. In view of this, sequential algorithms are restrictive for describing systems as they hide possibilities for doing things in parallel, and flexible algorithms are a better choice.
The task of designing (large) systems is complex and it is usual to give an overall description of the system and its components. This makes the task of design easier. Let us now give an overall description of a blood test machine/system using a diagram and then using a flexible algorithm.

Diagrammatic description of a blood test machine
This simple blood test machine receives a sample of blood, measures the amount of sugar and cholesterol in the blood by two independent tests and displays the results. To do this it splits the blood sample into two smaller samples.

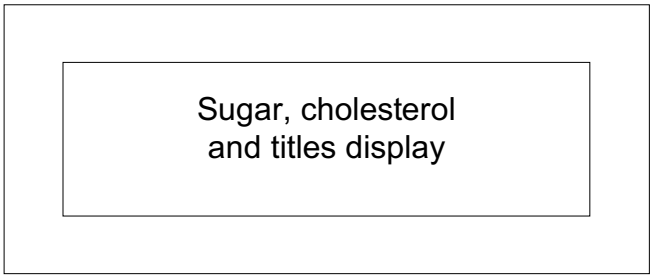


Description as a Flexible algorithm
The flexible algorithm description corresponds closely to the above diagram. It allows several designs for building such a machine. Here then is the flexible algorithm for the blood tester.

```
function     title1', sugar', title2', cholesterol' •=
bloodtester(blood_sample);
{ let sample1, sample2 •= split(blood_sample);
  title1'•="SUGAR";
  sugar' •= sugartester(sample1);
  title2'•="CHOLESTEROL";
  cholesterol' •= cholesteroltester(sample2); }
}; // end bloodtester
```

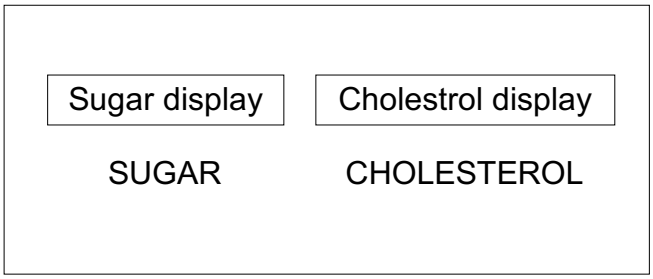
Here are three designs of the front panel of such a machine which are compatible with flexible execution. (Only the first design is compatible with sequential execution.)

First design of the front panel of such a machine



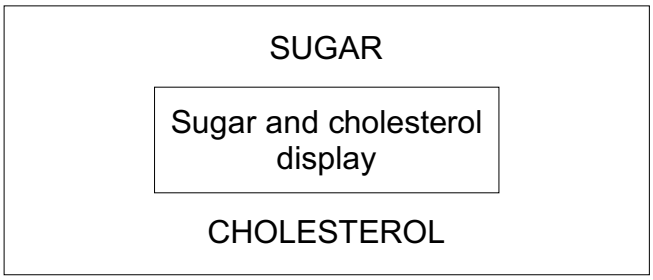
Here titles would be displayed when the algorithm is executed, and a larger (more expensive) display is needed. However, it is easy to adapt this design so that it can display the titles in more than one language.

Second design of the front panel of such a machine



Here the titles are "displayed" when the machine is built. Smaller (less expensive) displays are needed and perhaps less electricity is needed, which is important, particularly for a portable battery powered machine. However, it is not possible to adapt this design so that it can display the titles in more than one language.

Third design of the front panel of such a machine



This is a variation on the second design. It may be preferable to the previous design if the cost of a larger display is cheaper than the cost of two smaller displays, or if the larger display uses less electricity than two smaller displays.

PART 3 - A taste of things to come

Here we give a preview of certain topics which the student is likely to encounter as he progresses in his studies.

Why restrictions?
The restriction that there are only IN variables and OUT variables (i.e. no IN/OUT variables), means that there are no read/write conflicts. This is like a compile time check.
The restriction that assignment is once only, means that there are no write/write conflicts. This is like a run time check.
Thus, for a flexible algorithm with no errors these restrictions enable execution in parallel (and sequentially), giving unique results. They also encourage the writing of algorithms which may be executed in parallel (and sequentially). Thus, flexible algorithms may be viewed as parallel algorithms (as well as sequential algorithms).

Converting flexible algorithms to explicit parallel algorithms.
A flexible algorithm may be converted into a recursive parallel algorithm, using Dijkstra's *parbegin, parend* constructs. The form *{ statement_list }* is rewritten as *parbegin STATEMENT_LIST parend*, where *STATEMENT_*LIST is obtained from statement_list by replacing once only assignment by regular assignment. For a flexible algorithm complying with the restrictions above, the resulting parallel algorithm will give unique results. If it does not comply with these restrictions its behavior may be non-deterministic, giving different results for different executions on the same inputs. Also, given these restrictions, parallel evaluation of parameters in a function call is also possible.

Let us give an example using this conversion.

Consider the following fragment of a flexible algorithm using let blocks. Recall that in the following let, the variables i on the left and right hand sides of •= are different variables. Thus, there are three variables called i, and this allows the execution of the three statement lists to be overlapped and run in parallel as we shall show.

```
// Flexible algorithm style - Original version
{ let i•=5;
     < statement list 1 >
     { let i•=i+7 ;
          < statement list 2 >
          { let i•=i+1 ;
               < statement list 3 >
          }
     }
}
```

Since this in the style of a flexible algorithm, the statements may be reordered, and we can put the inner let block before the statement list as follows:

```
// Flexible algorithm style - Modified version
{ let i•=5;
    { let i•=i+7 ;
          { let i•=i+1 ;
               < statement list 3 >   // Innermost i used here.
          }
       < statement list 2 >   // Middle i used here.
    }
     < statement list 1 >   // Outermost i used here.
}
```

This is how this would look using the parbegin, parend constructs *where we use the convention in the following,* that the variables i on the left and right hand sides of := are different variables. Again there are three variables called i and this allows the execution of the three modified STATEMENT LISTs to be overlapped and run in parallel.

```
// Parallel algorithm style
parbegin let i:=5;
        parbegin let i:=i+7 ;
                parbegin let i:=i+1
                        < STATEMENT LIST 3 >   // Innermost i used here.
                parend
                < STATEMENT LIST 2 >   // Middle i used here.
        parend
          < STATEMENT LIST 1 >   // Outermost i used here.
parend
```

Note

Let us now return to the original flexible algorithm. In view of the restrictions we described earlier, the three variables i are IN variables and therefore cannot be assigned in the statement lists of the flexible algorithms. Thus, the fragment of the original flexible algorithm can also be written as a sequential algorithm, with only one variable i, using the assignment statement to update the value of i as follows.

```
// Sequential algorithm style
{ let i:=5;
     < STATEMENT LIST 1 >
     i:=i+7 ;
     < STATEMENT LIST 2 >
     i:=i+1 ;
     < STATEMENT LIST 3 >
}
```

The function addn rewritten

Here we have rewritten the definition using a labeled let block and have restructured it a little to allow more parallelism at the hardware level. If the flexible algorithm below were run on a multi-scalar processor or core, the two assignments on the line marked "//_**" would automatically be run in parallel. This is because the variables on the left and right hand sides of •= are different variables.

```
function s'•=addn(v,n)
// SPECIFICATION:v is a vector and n is its size which must be a power of 2.
//                    The function computes s' = v0 + v1 ... + vn-1
{
if    (n=1)
      {  s' •= v0}
         addn    (
                    n•=n/2;
                    loop:    {let i•=0;
                                  if  i < n
                                     { vi•=v2i + v2i+1; v(i+1)•=v2(i+1) + v2(i+1)+1;  //_**
                                       loop(i •= i+2)
                                     }
                              };
                 );
} // end addn
```

Embedded Flexible Language (EFL)

Subsequently to developing the course notes, the Jerusalem College of Technology has established a Flexible Computation Research Laboratory (FLEXCOMP lab). The laboratory staff and students have designed and implemented an Embedded Flexible Language (EFL) based on the Flexible Algorithms concept. Here is a link to the FLEXCOMP lab website http://flexcomp.jct.ac.il from where pre-compilers, publications and technical reports about EFL are freely available. At present educational material is not available for EFL, but the FLEXCOMP lab intends to develop such material.

Also, advanced constructs have been added to flexible algorithms. Explicit Bindings, Generalized Call Statements, and Composite assignments are described in [4]. A more sophisticated loop (logloop) is described in a technical report [5].

Here is an example of how the addn function described previously, may be written with the logloop construct of EFL:

```
EFL{
result = logloop( (1,2,3,4,-1,-2,-3,-4), add2);
}EFL
```

where add2(x, y) = x+y.

(The EFL pre-compiler considers + an operator, and not a function and thus + cannot be passed to logloop as a function object. Hence the need to define a function like add2 as above, which applies the operator + to the pair of parameters passed to it)

The parseq construct

Suppose that depending on a condition, we wish to execute a list of statements in parallel if the condition is true, or sequentially if the condition is false. We indicate this using the parseq keyword as follows.

```
parseq condition {statement list}
```

The parseq construct is equivalent to the following but is briefer to write:

```
if condition parbegin statement list parend
else {same statement list}
```

Suppose for example that the statement list processes a vector v, and parallel execution is inefficient on small vectors (e.g. vectors of length less than 32). It would be more efficient to reduce the parallelism as follows:

```
parseq length(v)<32 {statement list}
```

This construct affects the way the execution proceeds. With flexible algorithms, it does not affect the results produced.

References

1. R.B. Yehezkael, "Flexible Algorithms: Overview of a Beginners' Course", IEEE Distributed Systems Online, vol. 7, no. 11, 2006, art. no. 0611-oy002.
2. R.B. Yehezkael, "Flexible Algorithms: Fragments from a Beginners' Course", IEEE Distributed Systems Online, vol. 8, no. 2, 2007, art. no. 0702-o2001.
3. R.B. Yehezkael, "Flexible Algorithms-An Introduction", 2016, http://homedir.jct.ac.il/~rafi/flexalgo.pdf.
4. R. B. Yehezkael, M. Goldstein, D. Dayan and S. Mizrahi, "Flexible Algorithms: Enabling Well-defined Order-Independent Execution with an Imperative Programming Style", 4th Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), Brno, The Czech Republic, 2015. The most recent version of this paper is available at http://flexcomp.jct.ac.il/Publications/Flexalgo&Impl_1.pdf
5. M. Goldstein, D. Dayan, M. Rabin, D. Berlovitz, O. Berlovitz, R. B. Yehezkael, "EFL: An Embedded Language Enabling Well Defined Order-Independent Execution", Technical report, Flexible computation research laboratory (FLEXCOMP lab), Jerusalem college of technology, Jerusalem, Israel, 2015. Available at http://flexcomp.jct.ac.il/TechnicalReports/EFLprinciples.pdf
6. N. Francez, "*Program Verification*", Addison Wesley and Pearson Higher Education 1992
7. R.B. Yehezkael "Reasoning about Programs using Specifications and Induction", Internal Paper, Jerusalem College of Technology, revised 2004, available at http://homedir.jct.ac.il/~rafi/progver.pdf

Further information on this topic is available at http://homedir.jct.ac.il/~rafi/, in particular, see the section "Flexible Computation", and at http://flexcomp.jct.ac.il/.