Flexible Algorithms: Enabling Well-defined Order-Independent Execution with an Imperative Programming Style

Raphael. B. Yehezkael, Moshe Goldstein,
David Dayan and Shimon Mizrahi

Flexible Computation Research Laboratory,
Jerusalem College of Technology,
Jerusalem, Israel.

e-mail: flexcomp@jct.ac.il

November 2015 – כסלו תשע"ו

An earlier version of this paper appears in the proceedings of The 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015.

***Abstract*— Techniques are presented for ensuring well-defined parallel and unordered sequential execution (flexible execution), where values read are independent of the permitted execution orders. This is done by refining the scope rules of variables and defining where they may be initialized, where they may be updated, and where they may be read. Given these refined scope rules, this approach extends or replaces once-only assignment with suitable composite assignments to ensure well-defined read values. Examples of such suitable assignments are once-only assignment, "or=", "and=", "+=", "-=", etc.**

A flexible algorithmic language with these characteristics is described. The "core" of this language is based on functions with "IN", "OUT" but no "INOUT" parameters. The bodies of these functions are unordered sets of statements which may be executed in any order, but ensure well-defined values of functions.

Keywords*- *Flexible execution; Scope rules; Composite assignments

I. Introduction

Writing algorithms in a pure functional style enables well-defined parallel and unordered sequential execution and frees the programmer from considering all possible execution sequences. When writing parallel algorithms in an imperative programming language, the programmer is burdened to consider all possible execution sequences. We have designed a language, which on the one hand ensures well-defined parallel and unordered sequential execution, and on the other hand has an imperative programming style familiar to programmers.

We have also taken into account the need:

- For a notation to express well-defined parallelism so as to simplify its use.
- For a notation not too far from imperative programming languages.
- To educate/encourage programmers to write code which enables well-defined parallel execution if the full benefits of parallel execution are to be reaped.
- To have a notation where writing parallel algorithms is only slightly harder than writing sequential algorithms.
- To reduce the programming burden - using implicit parallelism rather than explicit parallelism.

We present our ideas via examples.

II. A Flexible Algorithm

Let’s give an example of a simple algorithm, written in the style of Python.

```
def vOut = reverse(vIn, low, high):
    # SPECIFICATION
    # IN – vIn is a vector
    #        low and high are positions within the vector vIn
    # OUT – If  low ≤ high, within the range [low … high],
    #            vOut is like vIn but reversed.
    #            Other elements of vOut are not given values
    #            by this function.
    #            If  low > high, then the function does nothing.
    if (low < high):
        vOut[high] = vIn[low]
        vOut = reverse(vIn, low+1, high-1)
        vOut[low] = vIn[high]
    elif (low == high):
        vOut[high] = vIn[high]
# end reverse
```

For example, to reverse a vector [1, 2, 3, 4, 5] and put the result in rOut we write:

```
rOut = reverse([1, 2, 3, 4, 5], 0,4)
```

Notes

1. Parameters may only be IN, which are given inside the parenthesis, or OUT which are given in the left-hand-side of the assignment-like prototype definition of the function. There may be several IN parameters and several OUT parameters.
2. It is an error to assign twice to the same simple OUT parameter or simple component of a multi-component OUT parameter.

III. Execution Methods

The above restrictions allow execution to be performed in a variety of orders, sequential or parallel, with *identical* end results. Execution methods are not described here. See the technical report [1] where the following execution methods are described.

a) Parallel shown in tree form.
b) Sequential using a stack, with immediate calls.
c) Sequential using a queue, with delayed calls.
d) Sequential using a current function data area, and a stack with delayed calls.

IV. Different ways of writing parameters in a function call

The definition of the function *reverse* given previously included a call written in functional style:

a) ***vOut = reverse (vIn, low+1, high-1)***

b) Sometimes the style of a procedure call is clearer:

reverse (vIn, low+1, high-1, vOut)

c) Sometimes an assignment-like style is helpful:

reverse (vIn=vIn, low=low+1, high=high-1, vOut=vOut)

d) ***This can be abbreviated showing only the changes:***

reverse (low=low+1, high=high-1)

Notes

1) Here we use styles (a) and (d) only.

2) There is a fundamental difference between low=low+1, high=high-1 as written in (d), and the assignment statement which updates values. Here the variables "low" and "high", on the left hand side, are new variables which will be created when the call is executed and the variables "low" and "high" on the right hand side, are existing variables holding values i.e. the variables on different sides of the assignment operator "=" are different variables. Though this may look like we are updating the values of variables, this most definitely is not the case. In fact, we are giving new variables their values when a function is called.

3) It is style (d) which gives our language an imperative style.

V. Convenient Extensions

The following extensions though not essential, are convenient, more readable, and briefer in many situations. We require that all such extensions can be converted to the core language described earlier. This ensures that flexibility of execution is retained.

A. Blocks and local variables

It is often useful to use a local variable to prevent re-computation of values. Suppose the value x+y-z is used several times in the definition of the function below.

```
def rOut = f(x, y, z):
    # SPECIFICATION …
    # statements of f
    … x+y-z …
    …
    … x+y-z …… x+y-z
    …
# end of  f
```

Then, to avoid re-computing this value, we can use a local variable in a block and write f as follows:

```
    def rOut = new_f(x, y, z):
    # Local variable(s) with their initial value.
    :(xyz = x+y-z)
        # statements of f with xyz in place of x+y-z
        … xyz …
        …
        … xyz …… xyz …
        …
# end of new_f
```

Note even though this is not written explicitly, the local variable of a block is of IN type only and is given a value with its declaration. This allows conversion to the core language and thus allows flexible execution.

B. Labeled blocks (or Loop Blocks)

We allow blocks to be labeled (really a convenient abbreviation for an auxiliary function definition and its call). This notation allows us to write in a style similar to loops ("for", "while"). We require the same scope rules for labels as for functions, variables and other names. Here is how we can square every element in a vector using a labeled block.

```
def vOut = squares(vIn):
    # SPECIFICATION
    # vIn, vOut are vectors having the same length.
    # Each element of vOut is the square of the
    # corresponding element of vIn.
    :loop (i = 0)    # vIn=vIn ; vOut=vOut
        if (i < vIn.length):
          loop(i = i+1)          # vIn=vIn ; vOut=vOut
                                 # for next "iteration"
          vOut[i] = vIn[i]**2
# end of squares
```

Note that "loop" is the name of the labeled block (or loop block) and "i" is its local variable. As with the block, the labeled block loop can be translated to the core language by using an auxiliary function as follows.

```
def vOut = new_squares(vIn):
    loop (i=0)    # vIn=vIn ; vOut=vOut
# end of new squares

def vOut = loop(vIn):
    if (i < vIn.length):
          loop(i = i+1)          # vIn=vIn ; vOut=vOut
                                 # for next "iteration"
      vOut[i] = vIn[i]**2
# end of loop
```

Again, this view treats a labeled block as a function which may have one or more external and internal calls. Again this addition will retain the flexible execution capability. In particular, parallel execution of several iterations of the loop is possible by using different local variables for each iteration.

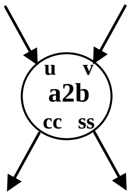
Note that for the purpose of definition, we have rewritten blocks and labeled blocks in terms of functions. This does not mean that we must implement them in this way.

More kinds of loops are described in a technical report [2].

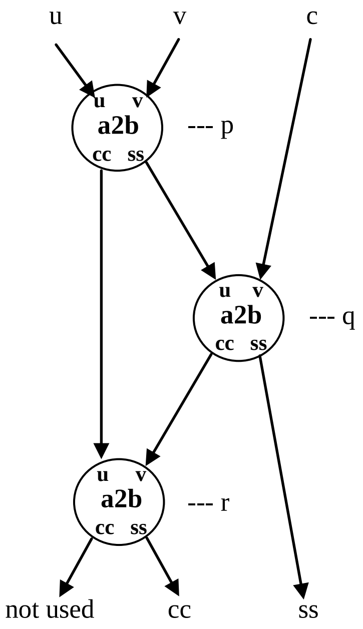
C. Explicit bindings

Explicit bindings are a direct way of showing how values pass between various functions. This is explained with an example, where the arrows show how values are passed; i.e. how arguments are bound to the appropriate formal parameters.

Let us assume that there is a hardware operation "a2b" for adding two bits giving their carry and sum respectively (i.e. a half adder). This operation can be represented by the following diagram.



Let us now construct a full adder "a3b" for adding three bits giving their carry and sum respectively. Here is a block diagram for "a3b" where for the sake of illustration, we only use the operation "a2b" in the diagram.



Note that because of the particular situation above, the last operation "a2b ---r" may be replaced by an "or", which is the usual way of making the construction.

The above block diagram can be expressed using the features we have so far described as follows.

```
def cc, ss = a3b(u, v, c):
    # SPECIFICATION
    # IN – u, v, c, are the bits to be added.
    # OUT – cc is the carry and ss is the sum.
    :(bit c1, s1 = a2b(u, v))
      :(bit c2, s2 = a2b(s1, c))
        _, cc = a2b(c1, c2)
        # the underscore on the left denotes an anonymous
        # variable, i.e. a not used result from a2b
        ss = s2
# end of a3b
```

This is not as direct as the diagram and there is a need for additional variables. To overcome this limitation of the textual form we use the brackets "[ ]" to make bindings explicitly.

```
def cc, ss = a3b(u, v, c):
    # SPECIFICATION
    # IN – u, v, c, are the bits to be added.
    # OUT – cc is the carry and ss is the sum.
    [
      a2b p, q, r    # like declaring three "objects"
                     # or "components" of "type" a2b.
      p.u = u  ;  p.v = v
      q.u = p.ss  ;  q.v = c
      r.u = p.cc  ;  r.v = q.cc
      cc = r.ss  ;  ss = q.ss
    ]
# end of a3b
```

Notes

1) In the function definition above, p, q, r correspond to the labels p, q, r in the diagram. We call a variable such as p.u a labeled input. We call a variable such as r.cc a labeled output.

2) All labeled inputs are required to receive a value but not all labeled outputs need be used. In the diagram and in the definitions of "a3b", the output r.cc from the third "a2b" is not used.

3) The use of the underscore "_" above is similar to its use in Prolog to indicate an anonymous variable. An anonymous variable may not be read, that is, it may not be used on the right hand side of an assignment statement. Each occurrence of "_" is considered to be a distinct anonymous variable.

4) Bindings in algorithms may not be nested. This is not restrictive as any number of operations may be bound together in a single binding.

5) Note that in assignments, labeled input variables occur on the left and labeled output variables occur on the right. Also, in this example we started with a directed acyclic graph, and expressed it in textual form using explicit bindings. It is also possible to start with the textual form of explicit bindings and produce a directed graph showing the bindings. We require the graph to be acyclic and consider it a syntax error if this is not the case. This prevents a loop occurring in the graph of the explicit binding, which is important for avoiding circular wait situations i.e. deadlock.

6) Explicit bindings can provide some of the mechanisms of objects, including a form of inheritance as well as omission of components. However, further work is needed to design an object mechanism compatible with flexible execution.

D. Generalized call statement and scope rules

The generalized call statement uses the assignment-like style to provide additional possibilities. This statement has three components: an entry block followed by the function to be called, followed by an exit block. Let's explain this by example.

Suppose we call a function f whose IN parameter is a vector v, and whose OUT parameter is a vector vOut. Suppose that we pass to f a vector of a hundred elements in which v[i] is i^2 and receive the result in aOut. A brief and clear way of doing this using the generalized call statement is as follows:

```
call
      :loop (i = 0)      # Entry block –
                         # Give IN parameters their value
        if (i < 100):
          loop (i = i+1)
          v[i] = i*i
      f                  # Call the function
      :aOut = vOut       # Exit block –
                         # OUT parameters transferred to
                         # their destination
# end of call
```

Here is something similar with OUT parameters. Suppose we wish to apply f to a vector "u" and put the first fifty elements of f(u) in aOut and the last fifty elements in bOut. Here is how this can be done without declaring an auxiliary vector.

```
call
      :v = u             # Entry block –
                         # Give IN parameters their value

      f                  # Call the function

      :loop (i = 0)      # Exit block –
                         # OUT parameters transferred to
                         # their destination
        if (i < 50):
          loop (i = i+1)
          aOut[i] = vOut[i]
          bOut[i] = vOut[i+50]
# end of call
```

Scope rules for IN and OUT parameters are handled this way:

The entry block would be typically executed before the function call. The exit block would be typically executed after the function results are evaluated. Note that as this is a call, the IN formal parameters may only be assigned in the entry block and the OUT formal parameters may only be read in the exit block. (The OUT formal parameters are not accessible in the entry block and the IN formal parameters are not accessible in the exit block.) This is the opposite to the access allowed in the body of the function.

This notation avoids the need to declare and initialize an additional vector. It is also suggestive, in that the entry and exit blocks may be executed either by the process which executes f or by the process which makes the call.

Generalized call statements may be nested.

VI. A More Complex Example

The bottom up algorithm for merge sort is presented using the constructs introduced earlier. It sorts a vector by repeated merging. For simplicity we assume that the vector's length is a power of 2. For example, suppose we wish to sort an 8-elements vector [8,1,7,2,6,3,5,4].

We merge element pairs to obtain [1,8,2,7,3,6,4,5].

Now we have four sorted pairs; so, we merge two pairs 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.

Now assume that there is a function m2 with specification as follows.

```
def vOut = m2(vIn, size, place):
    # SPECIFICATION
    # vIn -  a vector having two consecutive runs which
    #           are sorted in non-decreasing order.
    # These runs are of length "size" and at a position
    # "place" in vIn.
    # These two runs are merged into a single run of length
    # "2*size" which is put at position "place in vOut.
```

Let us use function m2 to define the function mergesort.

```
def vOut = mergesort(vIn):
    # SPECIFICATION
    # vIn, vOut  - two vectors having the same length,
    #                   which must be a power of 2.
    # vOut will be like vIn but sorted in non-decreasing
    # order.
    :loop (size =1, number = vIn.length)
      # the variable "size" is the size of the sorted run.
      # the variable "number" is the number of sorted runs.
      if (size == vIn.length):
        vOut = vIn
      else:
        loop(size=size*2
               number=number/2
               for (i=0; i < number; i=i+2):
                   vIn = m2(vIn, size, i*size)
             )
```

VII. Flexible Execution - Two Forms

Flexible execution takes two forms, parallel execution or unordered sequential execution. In both cases, function values are uniquely defined. Here we show how local variables may be used to allow more parallelism and how using suitable composite assignments enables unordered sequential execution.

A. *Introducing local variables to enable parallelism*

Consider the following fragment of a *sequential* program in Python style.

```
i = 5
…      # statements 1
…
i = i+7
…      # statements 2
…
i = i+1
…      # statements 3
```

In converting this to our flexible language we could write

```
:(i = 5)   # value of the first variable i is 5
  …        # statements 1 suitably modified to
  …        # single assignment form
  :(i=i+7)  # value of the second variable i is 12
    … # statements 2 suitably modified
    :(i=i+1)  # value of the third variable is 16.
      … # statements 3 suitably modified
```

Thus there are three variables called "i" and these allow the execution of the three modified statement lists to be overlapped and executed in parallel. A similar approach can be used with labeled blocks (loops) so as to allow several "iterations" to be overlapped and executed in parallel.

In the same spirit, new variables are created for parameters when a function call is executed, also enabling parallelism.

B. *Unordered sequential execution using composite assignments with a single function*

Languages such as "C" or "Java" [3, 4] allow composite assignments of the form "x f= y", meaning x=(x f y), where f is a binary operator or function of two arguments. A typical example is x += y. It also supports abbreviated forms of composite assignments such as x++ meaning x += 1 or x=(x + 1). Can composite assignments (and their abbreviations) be incorporated while retaining well-defined unordered sequential execution? For which kinds of functions and situations is this possible? Solutions to these questions are important as they will allow programmers to use the familiar algorithmic style, and they will enable flexible execution with reduced storage requirements. (The approach taken here differs from the approach of the previous subsection, in that we wish to allow flexible execution for certain composite assignments, *without* the need for multiple copies of a variable.)

For example, consider the following function:

```
def xOut = g(…)  xOut=$e_0$ :
    # initial value for xOut is $e_0$
        xOut f= $e_1$
        …
        xOut f= $e_n$
```

Here e_0, ..., e_n are expressions and f is as above. As in regular assignments, the IN and OUT restrictions must be observed in composite assignments. The variable on the left hand side of the assignment must be an OUT variable and the variables on the right hand side of the assignment must be IN variables.

Here are some conditions, which enable considerable execution flexibility while ensuring a uniquely defined final result for xOut.

1) It is required that ((a f b) f c) = ((a f c) f b). This is called the function condition for Unordered Sequential Composite Assignment Execution (function condition for USCAE). While the first parameter and the value of "f" must be of the same type, no restriction is placed on the second parameter of "f". Examples of functions satisfying this condition are "and", "or", exact arithmetic operations +, -, *, /, **, addition and subtraction with respect to a fixed modulus, etc. (Associative/commutative conditions or the existence of a unit element are not required.)

2) In this specific case, e_0, ..., e_n may be evaluated in any order, sequentially or in parallel. (In other situations it may be necessary to evaluate e_0 first.)

3) The initialization xOut= value of e_0, must be carried out before the composite assignments.

4) The composite assignments xOut f= value of e_1; ...; xOut f= value of e_n; must be performed in any sequential order.

5) Care must be taken that only the final value of an output variable is read or used in subsequent computations.

For now we assume that only one kind of assignment can be used with each variable. In the next subsection, this condition is relaxed.

The final value of xOut, equals value of the expression $(...(((e_0 \text{ f } e_{i_1}) \text{ f } e_{i_2}) \text{ f } e_{i_3}) \text{ ... f } e_{i_n})$ in an execution according to the above. Here i_1, i_2, ... i_n are a permutation of 1, 2, ... n and are determined by the order in which the composite assignments are performed.

Condition (1) the USCAE function condition, allows consecutive e's in the aforesaid expression, except for e_0, to be swapped, without altering the value of the expression. As it is possible to sort by swapping consecutive elements, we can rearrange e_{i_1}, e_{i_2}, ... e_{i_n} to e_1, e_2, ... e_n by sorting on the subscripts of the e's using "bubble sort" for example, there will be no change in the value of this expression by this sorting. So this means that the final value equals the value of $(...(((e_0 \text{ f } e_1) \text{ f } e_2) \text{ f } e_3) \text{ ... f } e_n)$ and so is uniquely defined.

Furthermore, we shall show that the USCAE function condition cannot be relaxed any further. Suppose that there are only two composite assignments and the final result is uniquely defined. If the first is made before the second, then

the final result will be ((e_0 f e_1) f e_2). But if the second is made before the first the final result will be ((e_0 f e_2) f e_1).
As the final result is uniquely defined, it follows that ((e_0 f e_1) f e_2) = ((e_0 f e_2) f e_1), which is the USCAE function condition.
(Incidentally, conditions similar to the above apply for detecting violations of once only assignments during a parallel execution. Expression evaluation may be in parallel but assignments to a given variable must be performed in any sequential order so as to guarantee detection of such violations. In a sequential implementation, this difficulty does not arise.)

C. Unordered sequential execution using composite assignments with several functions

Consider a more general case:

```
def xOut = g(…)  xOut=e0 :
    # initial value for xOut is e0
        xOut f1= e1
        …
        xOut fn = en
```

where f_1,...f_n are functions, which may or may not be different. (They may even all be the same, which is the case discussed above.) The USCAE function condition needs to be modified as follows:
1) It is required that ((a f_i b) f_j c) = ((a f_j c) f_i b) for $i \neq j$.
A similar argument to the above shows that the final value of xOut equals the value of (...(((e_0 f_1 e_1) f_2 e_2) f_3 e_3) ... f_n e_n) and is uniquely defined, subject to the other conditions enabling flexible execution. (In the sorting on the subscripts described above, the only change is to swap consecutive function expression pairs in the sort process and not just consecutive expressions.)

D. Hardware support for flexible execution

In the previous section, we saw that certain composite assignments give well-defined read values when executed in any sequential order. Are there such composite assignments which are easily implementable in hardware and give well defined read values? Are there such assignments which give well defined read values when executed on suitable hardware, in parallel or sequentially in any order?
As we know, dynamic memory is capacitor based. The advantage of this type of memory is its very high density. The disadvantage of this type of memory is its need for a refresh, and because of the refresh it has a lower speed. A significant advantage of this kind of memory, is that it facilitates <u>parallel and unordered sequential execution</u> when used with suitable composite assignments. For more information on dynamic memory see [5].
The simplest composite assignments which can be implemented with capacitor based memory are "or=" when capacitor empty represents 0, capacitor full represents 1. Similarly, "and=" can be implemented using capacitor full to represent 0, and capacitor empty to represent 1. These composite assignments may be executed in any sequential order or even in parallel. The reason why these composite assignments give well defined read values, even when executed in parallel, is because these operations would be performed at the capacitors themselves (and because of the electrical properties of the semi-conductors in the electrical circuits). Therefore, there is <u>no need to read the data stored in the capacitors</u>. It is interesting to ask if there are any other composite assignments which may be performed in parallel or in any sequential order, giving well defined read values for suitable kinds of hardware. These hardware aspects of composite assignments need further development and we intend to deal with this matter elsewhere.

E. Allowing any composite assignment by restricting flexibility

We can allow the use of any composite assignment by restricting the flexibility. For example, the expressions on the right hand sides of the above assignment statements may be executed in parallel <u>but</u> the composite assignments would be executed sequentially <u>in the order they are written</u>. This ensures well-defined read values providing all composite assignments to a variable are executed before its value is read. This is an interesting possibility as it provides a simple interface between sequential and parallel execution. Further study of this topic is needed, in particular, the efficiency of an implementation.

(Note that regular assignment is a special case of composite assignment. Defining (x f y) = y, makes x f= y the same as x=y. Thus our remarks here apply to regular assignment too.)

F. Benevolent side effects

Herlihy and Shavit [5] discuss benevolent side effects. In our discussion of composite assignments above, we indicated that certain composite assignments to the same variable would be performed in any sequential order.

If a function in an imperative programming language updates a global variable only with such assignments, read values will be well-defined providing that care is taken so that all updates complete before performing a read. Unordered well-defined sequential execution of the composite assignments is possible. This is like a benevolent side effect.

VIII. An Embedded Flexible Language (EFL)

We have implemented our ideas about flexible execution in the form of an Embedded Flexible Language (EFL) [2, 7]. Well-defined flexible execution order is implemented in EFL blocks embedded in a sequential host programming language. This approach allows the programmer to use sequential or well-defined parallel execution as appropriate and thus avoid the inefficiencies of excessive parallelism.

EFL's approach to parallel programming is as follows: in each program unit, (a) the *sequential* parts of the code (including all the Input/Output) are written in the host language, and (b) the *parallel* parts of the code are written inside embedded EFL language blocks. In these EFL blocks, order-independent execution is enabled by scope rules enforced inside the EFL blocks and the following restrictions:

(a) Only "*pure*" function calls may be used by the programmer. A *"pure"* function has no side-effects.

(b) There are *In* variables and *Out* variables (but no *InOut* variables).

(c) In EFL blocks, only *Once-only* assignments may be used.

EFL code exists within the context of the host language, with full access to the host program variables, context, and scope. Thus it has the memory overhead of the host language.

IX. Survey Of Previous Work

We have presented an algorithmic language which supports flexible execution. Blocks declarations and conditionals were written in the style of the Python language [8]. It differs from conventional algorithmic languages in that variables are either IN or OUT but not INOUT. Suitable composite assignments where the function satisfies the USCAE condition may be used, of which once-only assignment is a special case. On the other hand, pure functional languages [9, 10] completely avoid the assignment statement. If we were to write a program for reversing a vector in a functional language, at each swap a new vector would be created causing gross inefficiency. As was demonstrated above, in our approach, only one new vector was created. It also differs from functional languages in that they are higher level languages which make more use of higher order functions. Our language, on the other hand, is more algorithmic in style.

An early work by J.L.W. Kessels [11] presents a design of a language having both sequential and nonprocedural (parallel) blocks, where no assignments to global variables may be made. In our approach, well-defined access to global variables is provided via enhanced scope rules and suitable composite assignments.

In LUCID, [12] a well-formed program cannot cause a multiple assignment, which is checked at compile-time check; in our language it is a run-time check.

Our language handles multiple assignments at run time. Unlike the "pH" language, [13] our language does not have mutable structures (M-structures) but suitable composite assignments may be used for certain kind of OUT variables updates.

Thornley [14] presents a declarative language based on the ADA programming language incorporating parallel composition, parallel *for* loops, and single assignment variables for synchronization. This language has an algorithmic style but it does not include as general a construct as labelled blocks (for generalized looping). Also, it does not make use of composite assignments, or the assignment style for parameter passing, or the generalized call statement that we use in our language.

Vishkin [15] describes a comprehensive approach to parallel programming. It is based on the Parallel Random Access Machine (PRAM) model for describing algorithms and eXplicit Multi-Threading (XMT) approach for programming. The programmer is responsible for converting PRAM to XMT and a Work Depth (WD) methodology is described to aid the programmer do this task. A hardware implementation of PRAM is described too. It differs from our approach in that there are no scope rules or use of suitable composite assignments to ensure a well-defined result, this being the responsibility of the programmer. Our approach is based on implicit parallelism instead.

X. Conclusions

Achieving well-defined final values independent of execution order, depends on the following:

- IN parameters are assigned when calling a function, and may be read in the function's body.
- OUT parameters may be assigned in the body of a function, and their values may be read from their destination variables after exiting the function.
- There are no IN/OUT parameters.
- The function condition for Unordered Sequential Composite Assignment Execution (function condition for USCAE) enables unordered sequential execution of composite assignments to a given variable and ensures a well-defined final value.
- By restricting the flexibility, any composite assignment may be used and the final value of a variable is well-defined. The expressions on the right hand sides of assignment statements would be executed in parallel but the composite assignments themselves would be executed sequentially based on their textual order.

To sum up, our language enables well-defined flexible execution and has an algorithmic style familiar to programmers and engineers.

Acknowledgment

Sincere thanks to J.J. Florentin for his invaluable assistance and to Maurice Herlihy who made us aware of benevolent side effects.

References

[1] R.B. Yehezkael, M. Goldstein, D. Dayan and S. Mizrahi, "Flexible Algorithms and their Implementation: Enabling Well-defined Order-Independent Execution", 2013. Available at http://flexcomp.jct.ac.il/TechnicalReports/Flexalgo&Impl_1_full.pdf

[2] M. Goldstein, D. Dayan, M. Rabin, D. Berlovitz, O. Berlovitz and R. B. Yehezkael, "EFL: An Embedded Language Enabling Well Defined Order-Independent Execution", 2015. Available at http://flexcomp.jct.ac.il/TechnicalReports/EFLprinciples.pdf

[3] B.W. Kernigham and D.M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall 1988.

[4] J. Gosling, B. Joy, G. L. Steele, G. Bracha and A. Buckley, *The Java Language Specification – Java SE 7th Edition*, Addison Wesley 2013.

[5] J. Poulton, "An Embedded DRAM for CMOS ASICs", Proceedings of the 17th Conference on Advanced Research in VLSI, IEEE Computer Society Press, Sept 15-16, 1997, pp. 288-302. PDF version of paper available at http://www.cs.unc.edu/~jp/DRAM.pdf

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[7] D. Dayan, M. Goldstein, M. Popovic, Sh. Mizrahi, M. Rabin, D. Berlovitz, O. Berlovitz, E. Bussani Levy, M. Naaman, M. Nagar, D. Soudry, R. B. Yehezkael, "EFL: Implementing and Testing an

Embedded Language Which Provides Safe and Efficient Parallel Execution", submitted to this conference, 2015.

[8] *The Python Language Reference*, Release 2.7 2011, http://docs.python.org/reference/index.html

[9] S. Eisenbach, *Functional Programming: Languages Tools and Architectures*, Halsted Press: A division of John Wiley, 1987.

[10] B. O'Sullivan, J. Goerzen and D. Stewart, *Real World Haskell*, O'Reilly Media, 2009.

[11] J.L.W. Kessels, "A Conceptual Framework for a Nonprocedural Programming Language", *CACM* vol. 20, no. 12, 1977, pp.906 - 913.

[12] W.W. Wadge and E.A. Ashcroft, *LUCID, the Dataflow Programming Language*, Academic Press, 1988.

[13] R.S. Nikhil et al., *pH Language Reference Manual*, *Version 1.0-preliminary*, MIT Computer Structures Group Memo 369, 31 January 1995.

[14] J. Thornley, "Declarative Ada: parallel dataflow programming in a familiar context", *CSC '95, Proc. 23rd annual conf. on Computer science*, ACM 1995, pp. 73-80.

[15] U. Vishkin, "Using Simple Abstraction to Reinvent Computing for Parallelism", *CACM*,vol. 54, no. 1, 2011, pp. 75-85.