Deriving a dynamic programming algorithm for batch scheduling with the Refinement Calculator

Extended Abstract

Írem Aktuğ[1,2], Halit Oğuztüzün[2]
[1] *Royal Institute of Technology(KTH), Department of Microelectronics and Information Technology Stockholm, Sweden*
[2] *Middle East Technical University, Department of Computer Engineering, Ankara, Turkey*

In the stepwise refinement approach, a high-level specification is refined stepwise into an implementation. It is an application of the weakest precondition semantics introduced by Dijkstra(Dijkstra, 1976). The program construction process is a transformational reasoning proof: the expression specifying the problem is transformed via a series of relation preserving steps to another expression, i.e. a program, with some desired property such as efficiency or executability.

The Refinement Calculator is a tool developed at TUCS, Finland (Butler *et al.*, 1997), (Butler *et al.*, 2003), based on the refinement calculus of Back and von Wright(Back and von Wright, 1998). It supports the application of refinement transformations and the proof of verification conditions introduced by individual refinement steps. The refinement calculus theory was written in the proof assistant HOL as a theory. The calculator provides a user-friendly interface which is comprised of options linked to functions implemented as a part of this theory.

In this study, the Refinement Calculator has been used to derive a quadratic dynamic programming algorithm for a batch scheduling problem. A formal specification of the batching problem has been formulated as a HOL expression to be fed to the calculator. This expression consists of a precondition in the form of an assertion and a postcondition in the form of a nondeterministic assignment. Refinement options of the tool were applied to this specification in order to transform it into an executable program. The resulting program is in the imperative programming language offered by the calculator.

In the batch scheduling problem, there is a queue of N jobs that are to be processed in the same order, on one machine. The queue must be partitioned into batches and a partition with the minimum-cost is sought. For each job j, the execution time (t_j) and the cost factor (f_j) are given. There is also a constant batch startup time, S. All the jobs in a batch are output at the same time, after all of them have been executed. To calculate the cost of one job, the time elapsed since the start of the first batch is multiplied with the cost factor of the job. The sum of the costs of the jobs in a batch gives the cost of the batch. Similarly, the overall cost of a partition is the sum of the costs of the individual batches.

Albers and Brucker (1993) reduce this problem to a shortest path problem: The edges represent the batches, and the length of an edge represents the cost of running the corresponding batch. Thus, each partition is represented by a path from the first job to the $(N+1)^{st}$, which is inserted as a placeholder.

Let F_j be the length of a shortest path from *j* to *N+1*, $F_j(k)$ be the length of a shortest path from *j* to *N+1* which contains *(j,k)* as the first edge, and c_{ij} be the length of the edge from *i* to *j*. Then they observe:

$$c_{ij} = (\sum_{v=i}^{N} f_v)(S + \sum_{v=i}^{j-1} t_v) \tag{1}$$

$$F_j(k) = c_{jk} + F_k \tag{2}$$

$$F_j = min\{F_j(k) | j < k \leq N + 1\} \tag{3}$$

The cost (F_1) of the path starting from the first node is the required result.

In our study, the algorithm suggested by these observations has been derived in two stages. In the first stage, the edge lengths (c_{ij}'s) are calculated using (1). The specification for this part requires that the c matrix entry, indexed by the first and last jobs of a batch, gives the cost of the batch:

```
program costeval
var times:(num) array; factors:(num) array; s:num; n:num;
c:((num) array) array.
{ ((asize c) = (n+1))  /\
  (!j. ((j < (n+1)) ==> ((asize (lookup c j)) = (n+2)))) };
c:= c'. (contains_batch_cost c' (lookup times) (lookup factors) n s)
```

The derivation begins by the introduction of an outer loop that fills in complete rows. Then an inner loop is introduced to implement the setting of the entries of a single row of the matrix. In the meanwhile, a variable(fsum) to hold the sum of cost factors ($\sum_{v=i}^{N} f_v$) and another one(tsum) to hold the sum of execution times ($\sum_{v=i}^{j-1} t_v$) are introduced and incremented by the outer and inner loops, respectively. The resulting program for this first part can be seen in figure 1.

The majority of the proofs that arise in this first part of the algorithm are about the effects created by the basic array functions like lookup and update. Therefore, they resemble each other. In addition, arithmetic properties of the sigma operator, which is intended to implement the summation operator $\sum$, are utilized.

```
var times:num array,factors:num array,s:num,n:num,c:num array array.
 |[var fsum:num | T .
   |[var k:num | T .
     fsum := 0;
     k := n;
     do k > 0 ->
           fsum := (lookup factors k) + fsum;
           |[var tsum:num | T .
             |[var m:num | T .
               tsum := 0;
               m := k + 1;
               do m <= (n + 1) ->
                     tsum := (lookup times (m - 1)) + tsum;
                     c := update c k (update (lookup c k) m
                                (batchcost k m (lookup times)
                                           (lookup factors) n s));
                     m := m + 1
               od
             ]|
           ]|;
           k := k - 1
     od;
   ]|
 ]|
```

Figure 1. First Stage of the Algorithm: Computing the costs of individual batches

The second stage computes, for each job *k*, a minimum costing partition of jobs from *k* to *N* and the corresponding cost. The desired partition is the one containing the jobs from 1 to *N* and the initial specification(shown below) indicates the existence of such a partition p embedded in the array next. Furthermore, the cost of this partition will be recorded in array costn.

```
program computeXtoNcost
var times:(num) array; factors:(num) array; s:num; n:num;
c:((num) array) array; next:(num) array; costn:(num) array.
{ (n >= 1) /\
(contains_batch_cost c (lookup times) (lookup factors) n s) /\
((asize next) = (n+1)) /\ ((asize costn) = (n+2)) };
next,costn := next', costn'.
(?p. (contains_partition next' 1 n p) /\
     (sub_min_cost_p  (lookup times) (lookup factors) 1 n s p) /\
     ( (lookup costn' 1) =
        (overallcost  (lookup times) (lookup factors) n s p) )  )
```

The minimum costing partition of jobs k to N is constructed using one of the partitions containing the jobs m to N, where $m > k$, with the addition of the initial batch:[k,m]. The outer loop is introduced to accomplish this task for each job k. The proof that reflects the dynamic programming core of the algorithm comes about with the introduction of this loop and is both the most challenging and revealing part of the derivation. It brings about the usage of properties of partitions, elucidates the process of producing a minimum costing partition out of a smaller one and comprises the shortest path proof.

The inner loop merely scans the higher numbered jobs (m), in order to determine the smaller partition that will be used in the production of the new partition. The cost of the batch [k,m] is added to the cost of the partition containing jobs m to N to give the cost of the candidate for the new partition. The job m that minimizes this function is sought in this loop. The main proof performed for the introduction of the inner loop is a more complex version of selecting the minimizing element for a function with a subarray as the domain. The body of this inner loop is refined further by the introduction of a conditional statement, the omission of some statements with a reduction to skip and the use of the matrix c for the costs of individual batches([k,m]). The final algorithm is presented in figure 2.

These two separately derived programs are then combined by sequential composition. The first program outputs an array that contains the individual batch costs, while the second one uses the entries of this array as edge lengths when computing the cost of a path. In turn, the second program produces a next-batch array which implicitly contains the resulting partition and another array which contains its cost. The final algorithm is $O(N^2)$.

The Refinement Calculator is a tool that aims to render the Refinement Calculus applicable by programming community. It features a focus mechanism which we have found particularly powerful. However, our derivation experience brought about various shortcomings of the calculator that can limit its applicability to complicated problems. First of all, complete isolation of the user from HOL is yet to be accomplished, enhancing its usability. Stability of the software under large proof obligations needs to be addressed in future releases. As for programming features, an enhancement of the calculator interface with means to achieve data abstraction (e.g. partition data type) seems to be necessary for practical applications. Finally, stronger support for domain theories like graph theory remains crucial.

Acknowledgment

The authors would like to thank Linas Laibinis for his continuous support throughout this study.

References

Albers S., Brucker P. *The complexity of one-machine batching problems*, Discrete Applied Mathematics 47,pp. 87-107, 1993

Back R.J., von Wright J. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag. 1998

Butler M.J., Grundy J., Langbacka T., Ruksenas R.,von Wright J. *The Refinement Calculator: Proof Support for Program Refinement* Proceedings of Formal Methods Pacific '97, Wellington, New Zealand, Springer-Verlag, July 1997

```
var times:num array,factors:num array,s:num,n:num,
c:num array array, next:num array,costn:num array.
{(n >= 1) /\
 (contains_batch_cost c (lookup times)
                          (lookup factors) n s) /\
 ((asize next) = (n+1)) /\ (((asize costn) = (n+2))};
 costn:= update costn (SUC n) 0;
 |[var k:num | T .
    k := n;
    do k > 0 ->
    |[var m:num | T .
       m,next,costn := k + 2, update next k (k + 1),
                       update costn k
                          (lookup (lookup c k) (k + 1));
         do m <= (SUC n) ->
            if (((lookup (lookup c k) m) +
                              (lookup costn m))
                 < (lookup costn k))
               then
                    costn := update costn k
                              ((lookup (lookup c k) m) +
                                     (lookup costn m));
                    next := update next k m
               else
                    skip
            fi;
            m := m + 1
          od
    ]|;
     k := k - 1
     od
 ]|
```

Figure 2. The Second Stage of the Algorithm: Computing partitions

Butler M.J., Langbacka T., Ruksenas R., Wright J. The Refinement Calculator Tutorial and Manual. Available upon request from the authors.
Dijkstra E.W. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice Hall. 1976
Laibinis L. *Mechanised Formal Reasoning About Modular Programs*, PhD Dissertation, TUCS, Finland, April 2000
Morgan C.C. *Programming from Specifications (2nd Edition)*. Prentice Hall, 1994

Address for correspondence:
Írem Aktuğ
Department of Microelectronics and Information technology
Royal Institute of Technology(KTH)
Stockholm, Sweden
irem@imit.kth.se
Halit Oğuztüzün
Department of Computer Engineering
Middle East Technical University
Ankara, 06531, Turkey
oguztuzn@ceng.metu.edu.tr