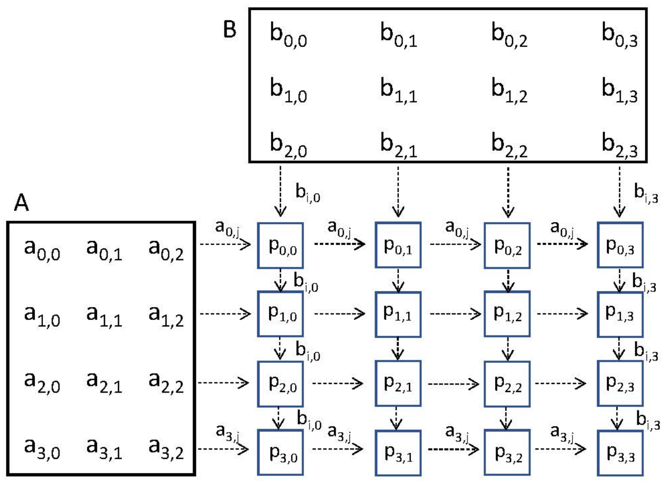
CISC 836 In-class Exercise: Take it Apart (2D-mesh-based Matrix Multiplication with UML-RT)

[The RSARTE model used in this exercise can be found here]

UML-RT capsules and connectors can be created at runtime using *optional capsules* and *unwired ports* that are wired at runtime using a mechanism similar to the popular *publish/subscribe* architectural style (as used, e.g., in MQTT, a version of HTTP for the Internet of Things). In short, a capsule `C1` can make a port `p1` accessible by registering `p1` as a *Service Provision Point (SPP)* using some string `str` (representing a 'topic'). Another capsule `C2` can then connect one of its ports `p2` to `p1` by registering `p2` as a *Service Access Point (SAP)* using `str`, assuming that `p1` and `p2` have the same protocol.

These features can be used to implement the multiplication of an m-by-n matrix A with an n-by-m matrix B using an algorithm that assumes a data flow architecture, i.e., a square grid of m-by-m processors (also known as a *2D mesh* or *systolic array*) that execute actions and produce output for other processors as the required input becomes available. The algorithm feeds the values in A into the grid in column-major order starting with the left-most column, while the values in B are provided in row-major order starting with the bottom row. For instance, for a 4-by-3 matrix A and a 3-by-4 matrix B, we get the following 4-by-4 processor grid:

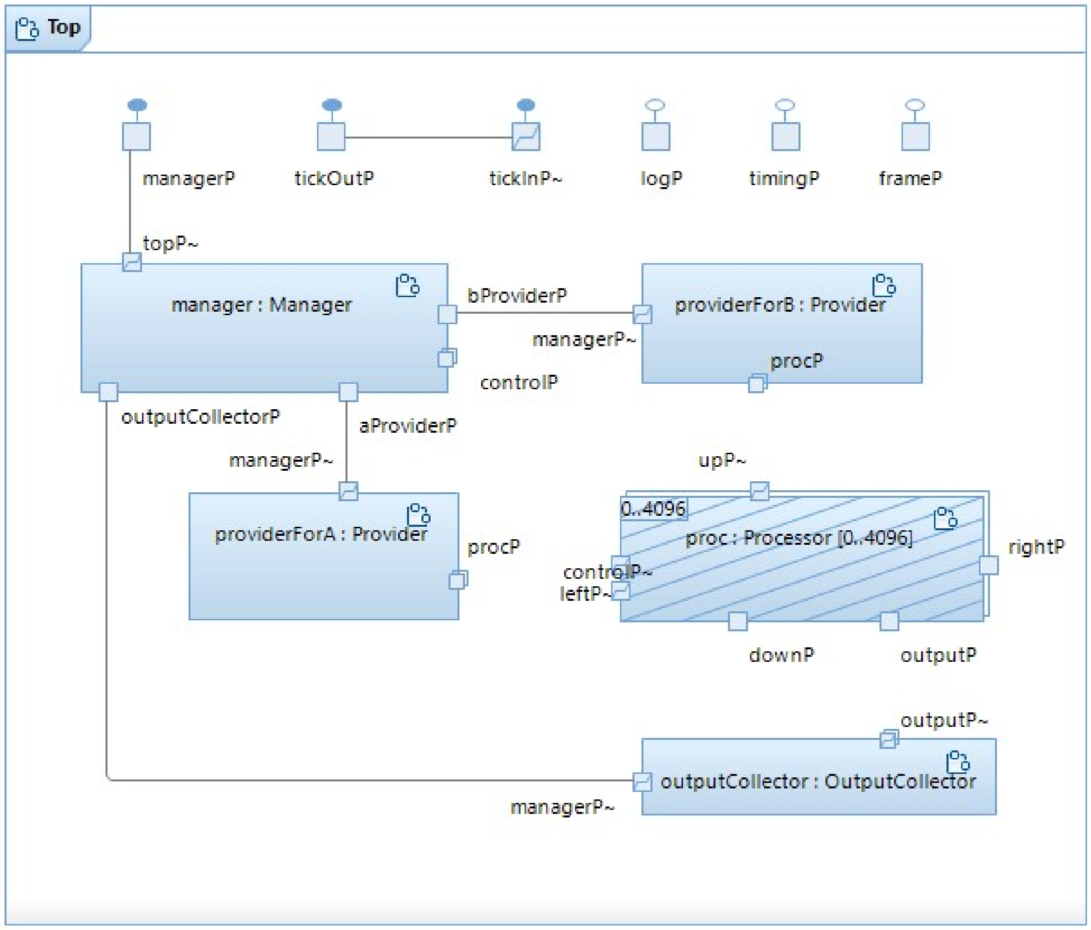


Once a processor in the grid has received a value from A from the left and a value from B from the top, it adds their product to a running total and also passes on these values to its right and down neighbours, respectively. I.e., each processor $p_{i,j}$ executes the following steps:

```
t = 0;
while (not all values received) {
    receive value $a$ from left neighbour (or $A$) and value $b$ from up neighbour (or $B$), in any order;
    pass on $a$ to right neighbour $p_{i+1,j}$ (if it exists);
    pass on $b$ to down neighbour $p_{i,j+1}$ (if it exists);
    $t := t + (a*b)$;
}
```

Once all values have been fed in, the running total t of processor $p_{i,j}$ constitutes the value $C_{i,j}$ in the product matrix. With m-by-m processors, the complexity of the algorithm is O(n). The algorithm is known as Canon's algorithm.

The algorithm is implemented in a UML-RT model with the following structure:



Each processor `proc` is an optional capsule instance that is dynamically created once the dimensions of the input matrices are known. The connectors between processors and two capsules that feed in the matrices are also created dynamically using SPP and SAP.

Capsule `Top`
Reads in matrix dimensions (*m* = #rows of first matrix *A* and *n* = #columns of *A*) and matrices *A* and *B*. Does some basic consistency checking. Sends dimensions to `manager`, then waits for confirmation that providers are ready for processors to be incarnated. Once that is received, it incarnates *m*m* processors (`proc[0]:Processor` through `proc[m*m-1]:Processor`), and then waits for confirmation that processors are ready to receive matrix values. Once that is received, it sends the matrix values to the `manager`. The sending alternates between the two matrices. For *B*, sending starts in the lower left corner *B[n,0]* and ends in the upper right corner *B[0,m]*, while for *A*, it starts in the upper right corner *A[0,n]* and ends in the lower left corner *A[m,0]*. E.g., for the matrices in the figure above the values would be sent as follows: $b_{2,0}$ $a_{0,2}$ $b_{2,1}$ $a_{1,2}$ $b_{2,2}$ $a_{2,2}$ $b_{2,3}$ $a_{3,2}$ $b_{1,0}$ $a_{0,1}$ $b_{1,1}$ $a_{1,1}$ … $b_{0,2}$ $a_{2,0}$ $b_{0,3}$ $a_{3,0}$.

Capsule part `manager:Manager`
After receiving the matrix dimensions from `Top` and passing them on to the providers and the `OutputCollector`, it prepares the dynamic connection to the processors (by setting up an SPP). It then sends a confirmation to `Top` that things are prepared for the incarnation of the processors and then waits for the processors to request the matrix dimensions. Once all processors have requested and been sent the dimensions, it waits till all processors have confirmed that they are ready to receive matrix values. Once that is received, it sends a `start` message to the processors, and confirms the readiness of the processors to `Top`. Then, it receives the values from `Top` and passes them on to the providers in an alternating fashion.

Capsule part `ProviderForA:Provider`
After receiving the dimensions from the `manager`, prepares for connecting with the m processors in the *left boundary* of the processor grid (i.e., `proc[i]` where `i%n==0`) (by setting up an SPP). Once all of them have sent connection requests, waits for values of matrix *A* and then feeds them to the processors.

Capsule part `ProviderForB:Provider`
As above, except that it connects with the *top boundary* processors of the processor grid (i.e., `proc[i]` with `0<=` `<=n-1`) and then feeds values from matrix *B* to these processors.

Capsule part `proc[i]:Processor`
Optional part incarnated by `Top` as required by the size of the input matrices. Will first connect with the `outputCollector` and the `manager` to receive the dimensions. Then, each processor will connect dynamically with its neighbours as required where
- processors in the *top boudary* of the grid connect with `providerForB` via their `up` port,
- processors in the *left boudary* of the grid connect with `providerForA` via their `left` port,
- processors in the *right boudary* of the grid will leave their `right` port unconnected,
- processors in the *bottom boundary* of the grid will leave their `down` port unconnected.
Once the connections are set up, receives the values and adds their product to a running total as described above. Once all values have been received, final value is sent to `outputCollector`.

Capsule part `outputCollector:OutputCollector`
Prepares the dynamic connection with the processors (by setting up an SPP). Waits for the dimensions from the `manager` and then collects and outputs all *m*m* values of the result matrix *C*.

A sequence diagram showing the possible message exchange for the multiplication of two 2-by-2 matrices is shown here.

Apart from optional capsuels and dynamically wired ports, the model also uses choice points, operations, data classes, port and capsule replication, defer/recall, optional capsules, and dynamic wiring. To allow for more of the algorithm steps to become visible on the level of the state machines, it also uses 'tick' messages that a capsule sends to itself to trigger the next transition.

Your task: What do you like, what do you not like about this model and the way it is implementing the multiplication?

Acknowledgement: The initial version of this model was created by Thomas Parker, CISC 836, Winter 2020

Last modified: Wed Feb 03 2021 16:16:28