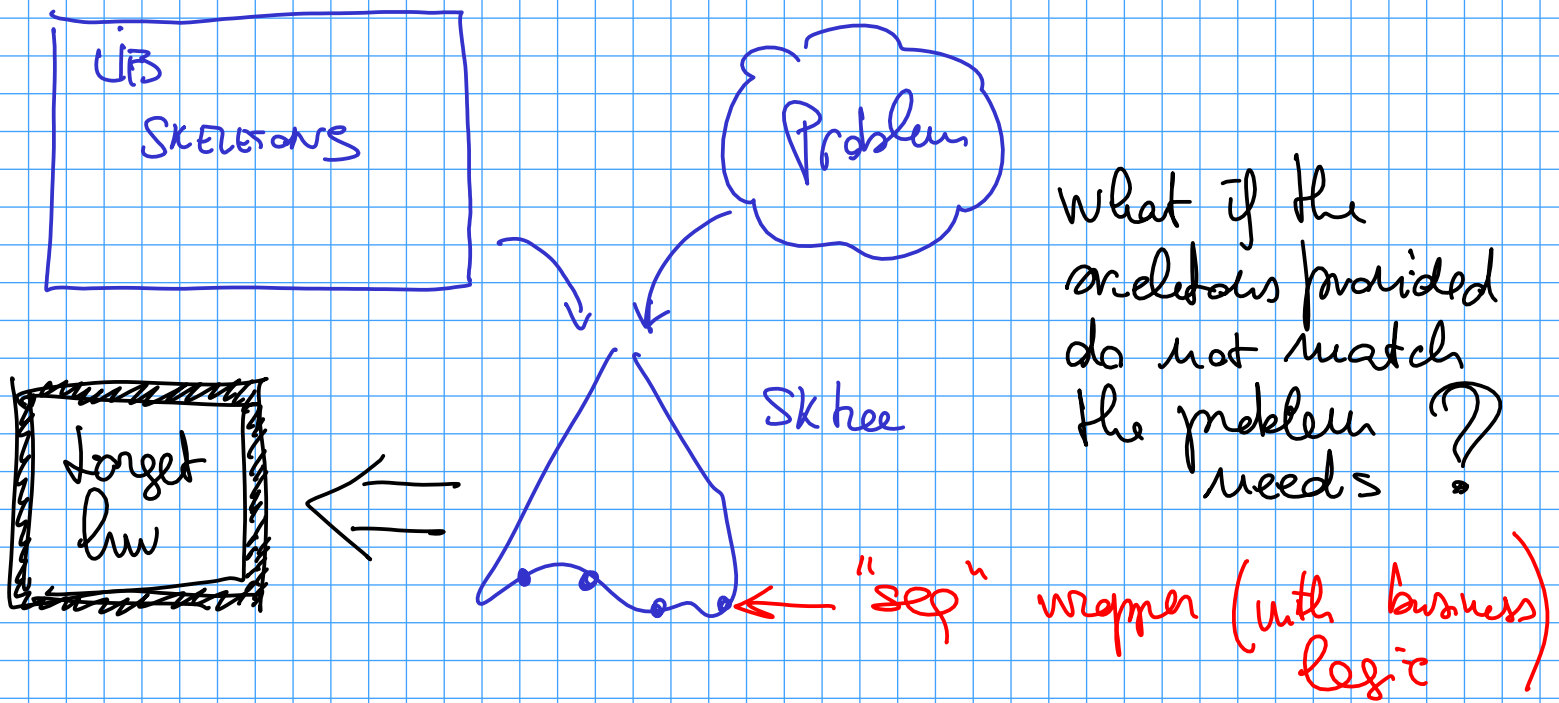
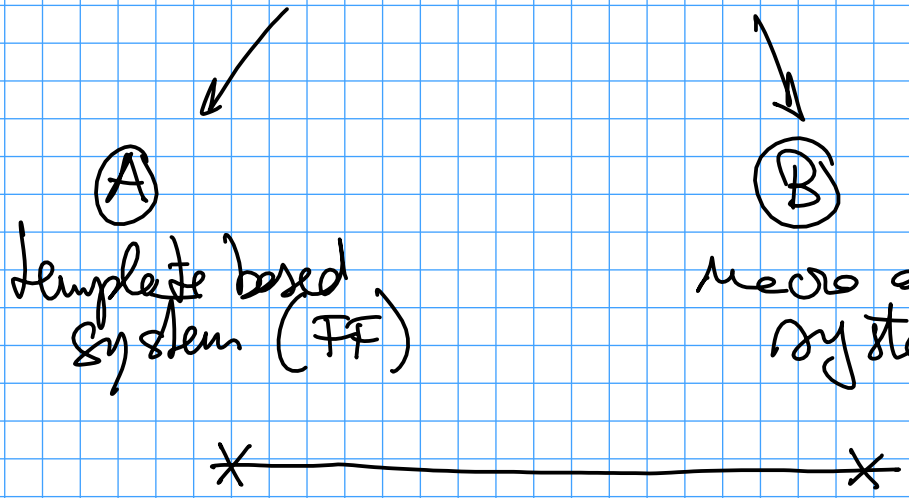


(1) how to get rid of the lib. limitations

(2) how can we be sure the extension do not impair the library features/results



Why do I need to expand a skeleton set?

→ common situations ⇒ need kind of support to deal with these situations

→ domain specific skeletons ⇒ need to be able to implement them

any other case : - probably you don't need a skeleton (as a general parallel pattern)

- you can probably consider a lower level implementation

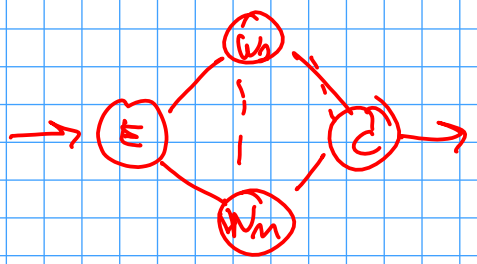
(A) try to solve question (1) and (2) above in a template based system.

FastFlow (pipe, farm, loopback, map, reduce)

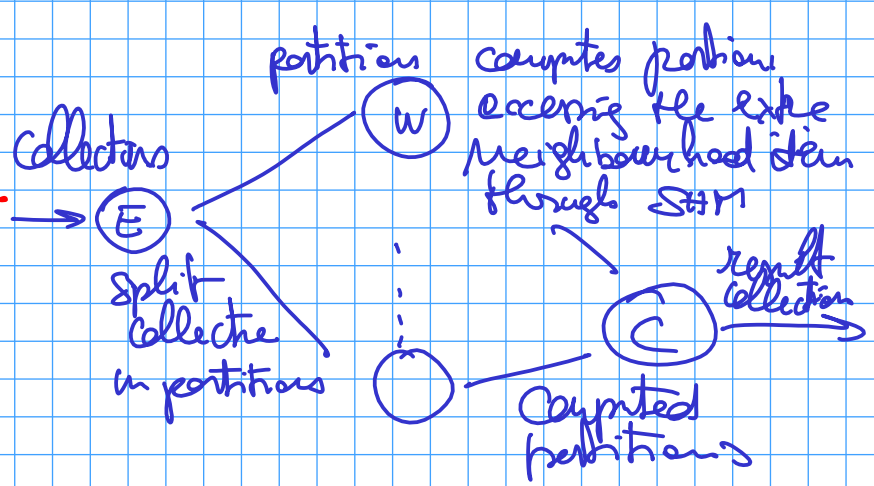
⟶ we need a stencil skeleton

Hp: the application programmer has access to the base component used to implement existing skels

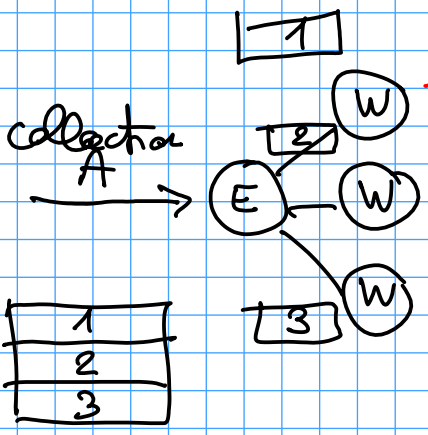
in such a way the appl. programmer may combine the components in different ways to program the stencil skeleton

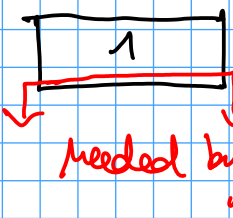


base component of FF (works with SHM)



What if workers commit the changes "in place"?





needed by "next" worker

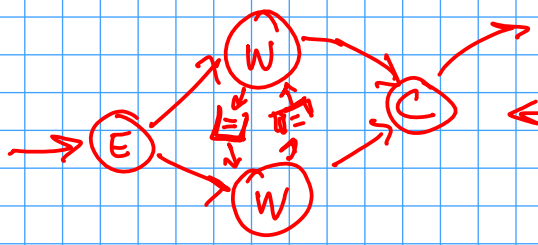
send a copy of this to "next" worker

then start working normally

1) counts new partition items
2) send A to the collector

because I don't have all the comun. channels needed

→ I must go down using the lower levels of FF implementation



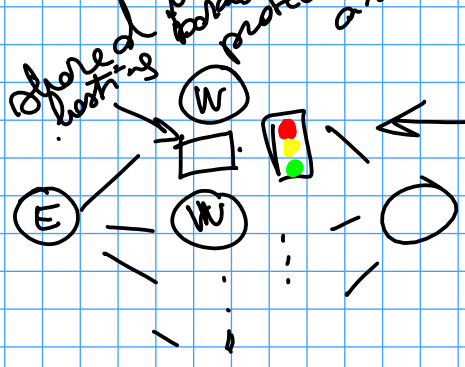
← implemented through the SPSC queues of FF

FARM PIPE ← APPL PROGRAMMER (in case you have all the skeletons needed)

MPSC SPMC

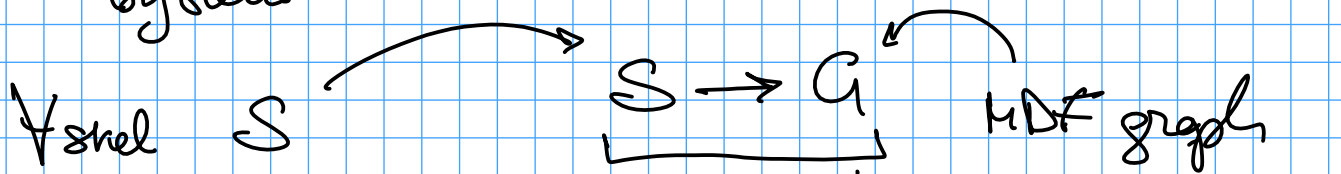
SPSC ← App programmer that need the additional channels in the template

App programmer with his own stencil stuff writes a new collection rather modifying the existing one

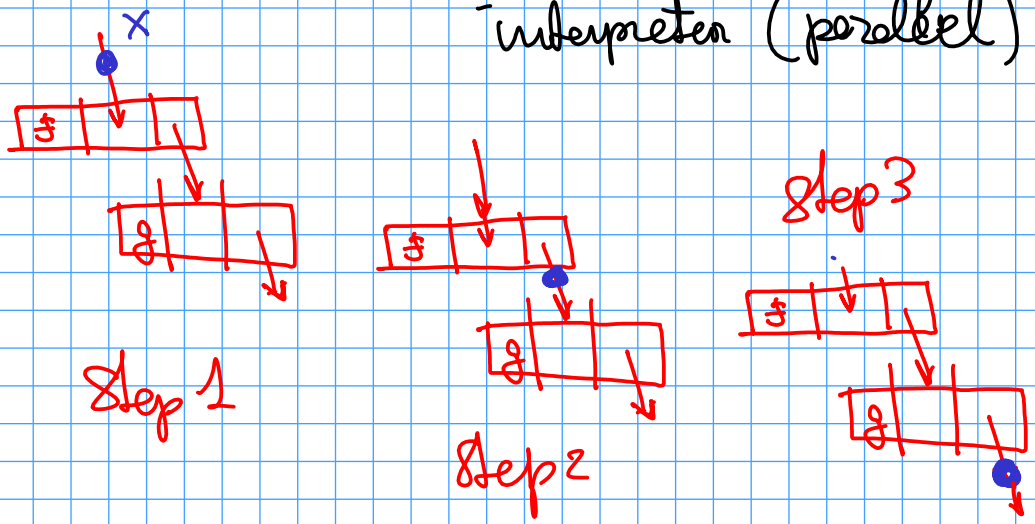


may seriously impair the decisions made by FF.

(B) try to solve question ① and ② above in a ~~template based system~~. Macro data flow system



pipe (f, g)



Skeletons

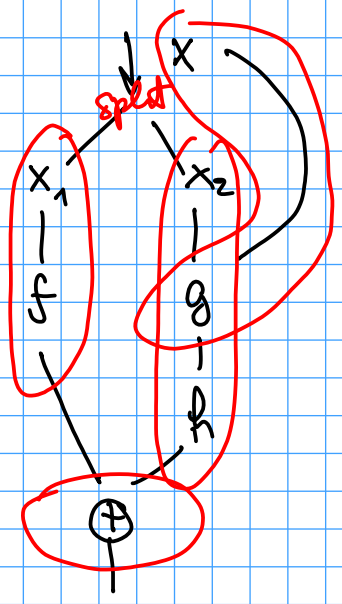
MDF graphs — expose the flow of data in the skeleton

MDF interpreter (executes any MDFg in the repository)

Muskel (Java & MDF)

muskel provided to the appl. programmer a MDF instruction abstraction

< identifier, Compute x, vector of input tokens, f vector of destinations, int >



<split_id, Split, [•], [<f_id, 1>, <g_id, 1>], ∅>

<f_id, F,

<g_id, G,

<h_id, H,

<⊕_id, Oplus, [•, •], [<output>], ∅>

```
MDFg x = new MDFg();
MDFi i_split = new MDFi( ↓ ↓ ↓ ↓ );
x.add_instruction(i_split);
;
```

```
Skeleton S = x.make_Skeleton();
```

Pros

- flow of data is known
- efficiency is guaranteed for decent grain MDFi

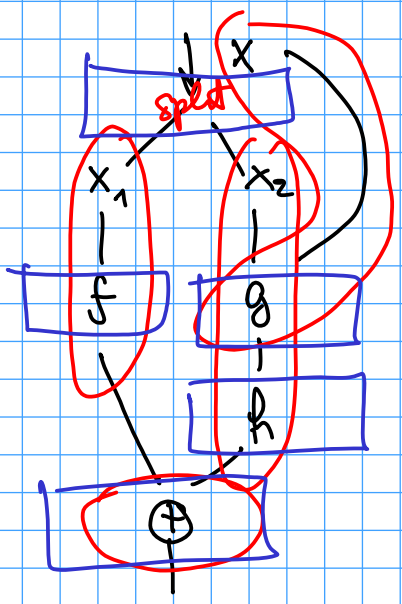
Cons

- it's relatively "hard" to program the MDFi and MDFg that way

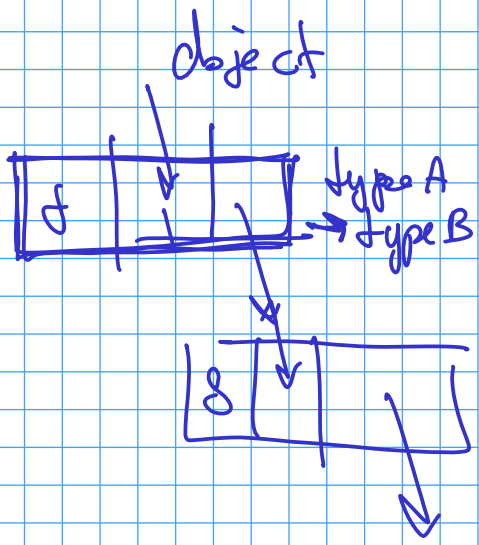
⇓

a solution?

GUI to draw the MDFg representing the skeleton




```
Skeleton mySS1
= nest
  mySS1(
    Skeleton f, g, h
    Merge  plus,
    Split  split
  );
```




```
Skeleton f = ...

Skeleton g = ...
```

```
MSFi<TypeA, TypeB> { ... }
```