CxSOM

Hervé Frezza-Buet

Herve.Frezza-Buet@centralesupelec.fr

Contents

1 Introduction

The `CxSOM` software suite enables to model consensus based multi-SOMs, explored in the BISCUIT team of teh Loria lab.

The current documentation is at work, and mainly technical stuff are reported here. See the README at the package root for an introduction, as well as examples.

2 The model

The computation enabled by `CxSOM` consists of successively updating data, keeping trace of the data values in history files.

2.1 Data instances (DI)

The elementary piece of data handled in `CxSOM` is called a *data instance* (*DI*). Each of the computed DIs ends as a record in a file, once it has been computed.

DIs are denoted by a triplet $[T, X, t]$, that reads as "the instance of variable X hosted by the timeline T at the time instant t of that specific timeline". Only elementary $[T, X, t]$ DIs matter, the concepts of *variable*, *timeline* and *time instant* only serve for describing the computation.

Each DIs is a value that is stored in a file once definitively computed. A value has a type, that can be:

- `Scalar`: A floating point value, usually in $[0, 1]$.
- `Pos1D`: A value in $[0, 1]$ corresponding to a position in a 1D map.
- `Pos2D`: A value in $[0, 1]^2$ corresponding to a position in a 2D map.
- `Array=`n: A value in $\mathbb{R}^n$, usually $[0, 1]^n$, $n \in \mathbb{N}^\star$.
- `Map1D<`$\mathcal{X}$`>=`n: A 1D map of type $\mathcal{X}$, i.e. a value in $\mathcal{X}^n$, $n \in \mathbb{N}^\star$.
- `Map2D<`$\mathcal{X}$`>=`n: A 2D squared map of type $\mathcal{X}$, i.e. a value in $(\mathcal{X}^n)^n$, $n \in \mathbb{N}^\star$.

where type $\mathcal{X} \in \{\texttt{Scalar}, \texttt{Pos1D}, \texttt{Pos2D}, \texttt{Array=}k\}$.

In `CxSOM`, the type is associated to a variable, i.e. all the $[T, X, \bullet]$ are DIs with the same type.

Every DI is also doted with a status variable $\langle [T, X, t] \rangle \in \{\texttt{busy}, \texttt{ready}\}$:

- `ready`: The computation of the value of the DI is definitively done. The DI value will not change anymore.
- `busy`: The definitive value of the DI is still to be determined.

More generally, in the following, $\langle x \rangle$ reads as "the status of x".

2.2 Updates

An *update* of some DI is the (re)computation by the `CxSOM` computer of the value of that DI. There are at most two updates for a DI, one for the first update (initialization) that is optional, and one, mandatory, for other updates.

2.2.1 An update is a function taking arguments and computing a result

An update u for a DI $[T, X, t]$ is made of:

- The decription of some computation, i.e. the function called to realize an update.

- The result res_u, i.e. $\text{res}_u \stackrel{\text{def}}{=} [T, X, t]$ itself, that is computed by the function of the update.

- Other DIs, thats serve as arguments to the function (their value is read when the function is called). Among arguments, a distinction is made between *in-arguments* and *out-arguments*:

    - The in-arguments in_u of the update u such as $\text{res}_u = [T, X, t]$ is the set of DIs used as arguments of u of the form $[T, \bullet, t]$. All the DIs handled in the simulation of the form $[T, \bullet, t]$ define a *time step* $\mathcal{S}_T^t$ of the simulation, so the in-arguments are the DIs used for the computation of an update which belong to the same timestep as the result.
    - The out-arguments out_u of the update u are the other DIs used as arguments for the computation of the update.

2.2.2 Datation and relaxation

Before it has a definitive value (i.e. before being in the `ready` status), the value of a DI may change several times. Indeed, all DIs in a timestep $\mathcal{S}_T^t = [T, \bullet, t]$ are updated until all of them get `ready`, and during that process, the values of the DIs may be recomputed several times. In order to handle the dependancies of the DIs within a timestep, we need to timestamp the values of the DIs, in order to determine those of them that have to be updated. Timestamp play the role of file dates in makefile, when the date of a target is compared to the date of its dependancies. Here, in order to know if an update u needs to be performed, we have to consider two things:

- Do we have $\forall [T, X, t] \in \text{out}_u$, $\langle [T, X, t] \rangle = $ `ready`? If not, the update cannot be done, it is considered as *impossible*.

- Have the in-arguments been updated since the last computation of the result? If the answer is yes, the result needs to be recomputed.

To compute the second conditon, every DI comes with a *datation* denoted by $\text{d}_{[T,X,t]} \in \mathbb{N}$.

An update then stores, for each of its in-arguments, the datation it had at the last computation of the result. When the simuator considers an update, it compares the current datation of the in-arguments to the ones stored by the update. If some are newer (or if the result has never been computed so far) the result is recomputed, and

$$\text{d}_{\text{res}_u} = 1 + \max_{[T,X,t]\in\text{in}_u} \text{d}_{[T,X,t]}$$

Considering an update for computation can thus lead to the following status of the update u, denoted by $\langle u \rangle$:

- $\langle u \rangle = $ `impossible`: $\langle \text{res}_u \rangle = $ `busy`, computation is not feasible yet, since some out-arguments are `busy`.

- $\langle u \rangle = $ `uptodate`: $\langle \text{res}_u \rangle = $ `busy`, nothing changed in the in-arguments input dates from last update, or the new value was not a significant modification. The datation d_{res_u} has not been modified.

- $\langle u \rangle = $ `updated`: $\langle \text{res}_u \rangle = $ `busy`, the computation has modified the value of res_u significantly.

- $\langle u \rangle = $ `done`: $\langle \text{res}_u \rangle = $ `ready`, the computation has modified the value res_u definitively.

- $\langle u \rangle = \texttt{none}$: Update status is not determined yet (used for initialization only).

The datation mechanism enables to control the update of all the DIs in a given timestep, i.e. all the DIs like $[T, \bullet, t]$ for the time instant t of the timeline T. When no more significant writes of the results can be done, the whole timestep is stable. So the relevance of the datation mechanism is to enable a *relaxation* of the DIs inside a timestep until stabilization of all of them is reached.

2.2.3 Update computation cycle

When a specific update is defined in `CxSOM` (as average, learning, etc.), the update is inherited from a base `cxsom::update::Base` class. In the subclass, the specificity of the computation has to be implemented, while the base class handles the details of the full update cycle, as datation and status. Algorithm 1 shows the cycle, the `on_*` calls are the method that subclass need to override in order to implement a specific update computation.

2.3 Timesteps and tasks

2.3.1 A computational structure for timesteps

In the simulator, the computation of the updates are put in a pool task and computed in parallel. However, since relaxation is related to the computation of updates belonging to the same timestep, this task management needs an intermediate level, which is a computational internal structure representing the timesteps.

Let us recall the notation $\mathcal{S}_T^t \overset{\text{def}}{=} \{[T', X', t'] \in \text{simulation} \mid T' = T \ \& \ t' = t\}$ for the timestep at instant t in the timeline T. For each $[T, X, t] \in \mathcal{S}_T^t$, the internal structure used in the simulator gathers at least an *usual update*, and an optional *initialization update*. The initialization update, if present, is used for the first feasible setting of the value of $[T, X, t]$ instead of the ususal update. Otherwise, the usual update is used throughout the successive setting of the value of $[T, X, t]$ during the relaxation.

In order to avoid further confusions, let us denote by $\overline{u}$ an update handled by a time step $\mathcal{S}_T^t$: it can be either a single update u (so u is the usual update) or a pair (u, u') if an initialization update is defined (then u' is the initialization update). In the case for which $\overline{u} = (u, u')$ is a pair, $\text{res}_u = \text{res}_{u'}$ obviously stands, so $\text{res}_{\overline{u}} \overset{\text{def}}{=} \text{res}_u$. In the case of a pair as well, the simulator uses u' until it gets a first computation of $\text{res}_{\overline{u}}$, and then it will switch to the use of u for next computations of $\text{res}_{\overline{u}}$.

As at least a call to the usual update u is required to be done for $\overline{u}$, we have particularized the status returned by initialization updates in algorithm 1, so that the evaluation of an initialization update can never lead to the definitive computation of res_u during the relaxation of the timestep.

2.3.2 Timestep status

The timestep structures $\mathcal{S}_T^t$ at the simulator level have a status

$$\left\langle \mathcal{S}_T^t \right\rangle \in \{\texttt{blocked}, \texttt{relaxing}, \texttt{checking}, \texttt{done}\}.$$

The status $\langle \mathcal{S}_T^t \rangle$ depends on the status of the updates of the timestep, observed when the updates are realized. The meaning of the $\langle \mathcal{S}_T^t \rangle$ is:

- $\langle \mathcal{S}_T^t \rangle = \texttt{blocked}$: The timestep is blocked due to impossible updates.
- $\langle \mathcal{S}_T^t \rangle = \texttt{relaxing}$: The timestep is under unstable computation.

4

Algorithm 1 `One update cycle for update` u

Require: a boolean attribute `out_ok`, telling if all out-args have been read successfully previously.
Require: a boolean attribute `is_init`, telling if the update is an initialization.

```
 1: if ⟨res_u⟩ = ready then
 2:    return done
 3: end if
 4: on_computation_start()
 5: if ¬out_ok then
 6:    // out-arguments need to be read.
 7:    for all [T, X, t] ∈ out_u do
 8:       if ⟨[T, X, t]⟩ = ready then
 9:          on_read_out_arg([T, X, t])
10:       else
11:          out_ok ← false
12:          on_read_out_arg_aborted()
13:          return impossible
14:       end if
15:    end for
16:    out_ok ← true // All in-args have been successfully read.
17: end if
18: for all [T, X, t] ∈ in_u do
19:    // Datation issues are handled in this loop, but not detailed here.
20:    on_read_in_arg([T, X, t])
21: end for
22: if none of the in-arguments have been updated since last cycle then
23:    if is_init then
24:       return updated // updates used as initialization are never considered as up-to-date.
25:    else
26:       return uptodate
27:    end if
28: end if
29: // From here, we know that the result has to be recomputed.
30: x = none // This is the status we plan to return, it is set next.
31: α = on_write_result(res_u) // α is a Boolean telling if the value change is significant.
32: if we have just written a new significant value in res_u then
33:    x ← updated
34: else
35:    if is_init then
36:       return x ← updated // Even not significant changes are considered as an actual update for
          initialization updates.
37:    else
38:       return x ← uptodate
39:    end if
40: end if
41: // Before returning, we have to check if in-arg may change in the future
42: if all in-arguments are ready then
43:    if is_init then
44:       x ← updated // As u an init, we return updated since further change can be consider by the
          usual update comming next for updating res_u.
45:    else
46:       x ← done // We have definitively computed the result...
47:       ⟨res_u⟩ ← ready // The result DI value is locked.
48:    end if
49: end if
50: return x
```

- $\langle \mathcal{S}_T^t \rangle = $ `checking`: Every update seem stable, we are checking this.

- $\langle \mathcal{S}_T^t \rangle = $ `done`: The timestep is done, all updates $\overline{u}$ have lead to $\mathrm{res}_{\overline{u}} = $ `ready` and have quit the simulator.

2.3.3 Timestep update queues

The timestep structures $\mathcal{S}_T^t$ at the simulator level are doted each with 4 update queues, where the updates $\overline{u}$ are stored. Each the $\overline{u} \in \mathcal{S}_T^t$ belongs to one of the 4 queues of $\mathcal{S}_T^t$.

When a timestep is asked by the simulator to provide computation, it extracts updates from some of the queues. When the update is performed by the simulator, it is given back to the timestep, with the return status resulting from algorithm 1. According to this status, the update is stored in the appropriate queue, and some supplementary transfers from one queue to another queue may happen. This will be detailed further, as the *timestep state machine.*

For now, let us present the 4 queues used by a timestep $\mathcal{S}_T^t$.

$$
\begin{aligned}
[\mathcal{S}_T^t]_{\text{unstable}} &= \{\overline{u} \in \mathcal{S}_T^t \mid \langle \overline{u} \rangle \text{ needs to be known.}\} \\
[\mathcal{S}_T^t]_{\text{impossible}} &= \{\overline{u} \in \mathcal{S}_T^t \mid \langle \overline{u} \rangle \text{ have been detected as } \texttt{impossible}.\} \\
[\mathcal{S}_T^t]_{\text{stable}} &= \{\overline{u} \in \mathcal{S}_T^t \mid \overline{u} \text{ have been seen stable for the first time are here.}\} \\
[\mathcal{S}_T^t]_{\text{confirmed}} &= \{\overline{u} \in \mathcal{S}_T^t \mid \overline{u} \text{ for which stability is confirmed.}\}
\end{aligned}
$$

2.3.4 Providing jobs

A timestep is periodically asked by the simulator to provide tasks, i.e. to offer updates that will be evaluated by the simulator. These updates are extracted from the timestep and inserted in a pool task, waiting for execution. The timestep keeps trace of their existency while they are outside, waiting for being executed.

So when a timestep $\mathcal{S}_T^t$ is asked for new updates to be done, it provides (and extracts) :

- all the $\overline{u} \in [\mathcal{S}_T^t]_{\text{unstable}} \cup [\mathcal{S}_T^t]_{\text{stable}}$ if $\langle \mathcal{S}_T^t \rangle \in \{\texttt{relaxing}, \texttt{checking}\}$

- nothing otherwise.

2.3.5 The timestep state machine

When an update is executed, it is given back to the timestep by the simulator, with a status report given by algorithm 1.

The reporting of an update makes the update to be inserted in a queue of the timesteps. This is done as detailed in algorithm 2.

2.3.6 Update patterns

Defining an update for all DIs (i.e. at each time t) would be exhausting. It can be done for specific time instants (usually $t = 0$), but a generic update definition is required for the general cases.

This is what *update patterns* (UP) are introduced for. Let us denote a *relative data instance* (RDI) as $\{T, X, \tau\}$, where $\tau \in \mathbb{Z}$ is a relative time shift. A RDI can be *anchored* as a DI at time t. This consists in transforming $\{T, X, \tau\}$ into $[T, X, t + \tau]$.

The definition is then similar to the one of an update, presented in section 2.2.1. An UP π is made of:

Algorithm 2 Reporting update $\overline{u}$ to the timestep $\mathcal{S}_T^t$

Require: $\langle\overline{u}\rangle$ is the status of the update execution.
Require: $\langle\mathcal{S}_T^t\rangle$ is the current status of the timestep.

 1: **if** $\langle\overline{u}\rangle = \texttt{impossible}$ **then**
 2:     $[\mathcal{S}_T^t]_{\text{impossible}} \leftarrow [\mathcal{S}_T^t]_{\text{impossible}} \cup \{\overline{u}\}$
 3: **else if** $\langle\overline{u}\rangle = \texttt{updated}$ **then**
 4:     $[\mathcal{S}_T^t]_{\text{unstable}} \leftarrow [\mathcal{S}_T^t]_{\text{unstable}} \cup \{\overline{u}\}$
 5: **else if** $\langle\overline{u}\rangle = \texttt{done}$ **then**
 6:     // The update is not handled anymore, since $\langle\text{res}_{\overline{u}}\rangle = \texttt{ready}$.
 7: **else if** $\langle\overline{u}\rangle = \texttt{uptodate}$ **then**
 8:     **if** $\langle\mathcal{S}_T^t\rangle = \texttt{checking}$ **and** $\overline{u}$ was in $[\mathcal{S}_T^t]_{\text{stable}}$ **then**
 9:        $[\mathcal{S}_T^t]_{\text{confirmed}} \leftarrow [\mathcal{S}_T^t]_{\text{confirmed}} \cup \{\overline{u}\}$
10:     **else**
11:        $[\mathcal{S}_T^t]_{\text{stable}} \leftarrow [\mathcal{S}_T^t]_{\text{stable}} \cup \{\overline{u}\}$
12:     **end if**
13: **end if**
14: $\texttt{update\_status}\left(\mathcal{S}_T^t\right)$ // See algorithm 3.

Algorithm 3 $\texttt{update\_status}\left(\mathcal{S}_T^t\right)$

 1: **if** $[\mathcal{S}_T^t]_{\text{impossible}} \neq \emptyset$ **then**
 2:     $[\mathcal{S}_T^t]_{\text{stable}} \leftarrow [\mathcal{S}_T^t]_{\text{stable}} \cup [\mathcal{S}_T^t]_{\text{confirmed}}$
 3:     $[\mathcal{S}_T^t]_{\text{confirmed}} \leftarrow \emptyset$ // Blocking out-arguments lead to reconsider the $\mathcal{S}_T^t$ stability.
 4:     **if** all the out-arguments of the $\overline{u} \in \mathcal{S}_T^t$ are $\texttt{ready}$ **then**
 5:        $[\mathcal{S}_T^t]_{\text{unstable}} \leftarrow [\mathcal{S}_T^t]_{\text{unstable}} \cup [\mathcal{S}_T^t]_{\text{impossible}}$
 6:        $[\mathcal{S}_T^t]_{\text{impossible}} \leftarrow \emptyset$ // Impossible updates become instable, i.e. evaluable.
 7:        $\langle\mathcal{S}_T^t\rangle \leftarrow \texttt{relaxing}$
 8:     **else**
 9:        $\langle\mathcal{S}_T^t\rangle \leftarrow \texttt{blocked}$
10:        // At this level, the simulator consider the timesteps $\mathcal{S}_{T'}^{t'}$ owning busy out-arguments as *blockers* of $\mathcal{S}_T^t$.
11:     **end if**
12: **else if** $[\mathcal{S}_T^t]_{\text{unstable}} \neq \emptyset$ **then**
13:     // $[\mathcal{S}_T^t]_{\text{impossible}} = \emptyset$
14:     $[\mathcal{S}_T^t]_{\text{stable}} \leftarrow [\mathcal{S}_T^t]_{\text{stable}} \cup [\mathcal{S}_T^t]_{\text{confirmed}}$
15:     $[\mathcal{S}_T^t]_{\text{confirmed}} \leftarrow \emptyset$ // A single unstable argument leads to reconsider the $\mathcal{S}_T^t$ stability.
16:     $\langle\mathcal{S}_T^t\rangle \leftarrow \texttt{relaxing}$
17: **else if** $[\mathcal{S}_T^t]_{\text{stable}} \neq \emptyset$ **then**
18:     // $[\mathcal{S}_T^t]_{\text{impossible}} = \emptyset, [\mathcal{S}_T^t]_{\text{unstable}} = \emptyset$
19:     $\langle\mathcal{S}_T^t\rangle \leftarrow \texttt{checking}$ // $\mathcal{S}_T^t$ updates are all stable, stability confirmation is in progress.
20: **else**
21:     // $[\mathcal{S}_T^t]_{\text{impossible}} = \emptyset, [\mathcal{S}_T^t]_{\text{unstable}} = \emptyset, [\mathcal{S}_T^t]_{\text{stable}} = \emptyset$
22:     // Every update stability is confrmed. We can set all the results as $\texttt{ready}$
23:     **for all** $\overline{u} \in [\mathcal{S}_T^t]_{\text{confirmed}}$ **do**
24:        $\langle\text{res}_{\overline{u}}\rangle \leftarrow \texttt{ready}$
25:     **end for**
26:     $\langle\mathcal{S}_T^t\rangle \leftarrow \texttt{done}$
27:     **for all** $\mathcal{S}_{T'}^{t'}$ for which $\mathcal{S}_T^t$ is a blocker **do**
28:        $\texttt{update\_status}\left(\mathcal{S}_{T'}^{t'}\right)$
29:     **end for**
30: **end if**

- The decription of some computation, i.e. the function called to realize an update.

- The result res_π, which is a zero-shift DRI $\{T, X, 0\}$.

- Other RDIs or DIs, thats serve as arguments to the function.

Anchoring an UP π such as $\text{res}_\pi = \{T, X, 0\}$ at time t consists in defining an update by anchoring the result and all the arguments which are RDIs, in order to get a regular update for $[T, X, t]$.

2.4   Simulation

The simulator compute DIs. It does it by asking tasks to its active timesteps, as explained in section 2.3.4. When it is out of work, it tries to generate more DIs to compute by anchoring available UPs, thus creating new active timesteps. This is described in algorithm 4.

Algorithm 4 `get_one_job` ()

Require: A task queue (jobs) denoted by $\mathcal{J}$, used and updated by the algorithm.

1: **if** $\mathcal{J} = \emptyset$ **then**
2:    // Let us try to fill the task queue $\mathcal{J}$.
3:    **for all** $\mathcal{S}_T^t$ which are handled currently **do**
4:       **if** $\langle \mathcal{S}_T^t \rangle \in \{\texttt{relaxing}, \texttt{checking}\}$ **then**
5:          `push` $\left(\mathcal{J}, \, [\mathcal{S}_T^t]_{\text{unstable}} \cup [\mathcal{S}_T^t]_{\text{stable}} \right)$ // see section 2.3.4
6:       **end if**
7:    **end for**
8:    **if** $\mathcal{J} = \emptyset$ **then**
9:       // No active patterns are able to provide new jobs.
10:       // We create new timesteps by anchoring the UPs.
11:       `anchor_UPs` () // See algorithm 5.
12:       // We may have new $\mathcal{S}_T^t$s, let us retry to ask them jobs.
13:       **for all** $\mathcal{S}_T^t$ which are handled currently **do**
14:          **if** $\langle \mathcal{S}_T^t \rangle \in \{\texttt{relaxing}, \texttt{checking}\}$ **then**
15:             `push` $\left(\mathcal{J}, \, [\mathcal{S}_T^t]_{\text{unstable}} \cup [\mathcal{S}_T^t]_{\text{stable}} \right)$ // see section 2.3.4
16:          **end if**
17:       **end for**
18:    **end if**
19: **end if**
20: **if** $\mathcal{J} = \emptyset$ **then**
21:    **return** // There is really no way to get a new job.
22: **else**
23:    $j \leftarrow$ `pop` $(\mathcal{J})$
24:    **return** j // That job will be done.
25: **end if**

---

3   Available update functions

3.1   Initializations

This sets the variable X so that each of the float it contains is 3.14, Y so that each of the float it contains is a random value in $[0, 1[$, Z as a copy of X.

```
"X" << fx::clear() | kwd::use("value", 3.14);
"Y" << fx::random();
"Z" << fx::copy("X");
```

Algorithm 5 `anchor_UPs` ()

Require: The set of defined UPs is referred to as $\mathcal{P}$.

1: Determine the used variables $\mathcal{V} \stackrel{\text{def}}{=} \{(T, X) \mid \pi \in \mathcal{P} \ \& \ \text{res}_\pi = \{T, X, \tau\}\}$.
2: For each timeline T mentionned in $\mathcal{V}$, compute @T as the first time instant for which a variable $(T, X) \in \mathcal{V}$ is busy (i.e. not computed so far).
3: **for all** $\pi \in \mathcal{P}$ **do**
4:    Consider $\{T, X, 0\} = \text{res}_\pi$
5:    **if** $\langle [T, X, @T] = \text{busy} \rangle$ **then**
6:       Anchor π at time instant @T, thus creating a new timestep in the simulator for the next instant to be computed in that timeline.
7:    **else**
8:       // The DI computed by π is already available at @T, some other UPs are late, they will be instanciated first.
9:    **end if**
10: **end for**

3.2   Matching a value against weights

In SOM-like computation, an input ξ is given and is matched against all the weights w_i in the map. The result is an activity vector, its elements a_i are in $[0, 1]$, corresponding to each weight, that is 1 if the weight and the input perfectly match.

The formula is, for a map of weights w_i and an input ξ, $\forall i, a_i = \mu(\xi, w_i)$. Two matcing functions are defined, a triangle matching $\mu_\triangle$ and a Gaussian matching μ_G.

$$\mu_G(\xi, w) \stackrel{\text{def}}{=} \exp\left(\frac{(\xi - w)^2}{2\sigma^2}\right), \ \mu_\triangle(\xi, w) \stackrel{\text{def}}{=} \max\left(1 - \frac{|\xi - w|}{r}, 0\right)$$

```
kwd::type("Xi",    "Pos2D"            );
kwd::type("Wgt",   "Map1D<Pos2D>=100" );
kwd::type("ActT",  "Map1D<Scalar>=100");
kwd::type("ActG",  "Map1D<Scalar>=100");

"ActT" << fx::match_triangle("Xi", "Wgt")   | kwd::use("r",     .3);
"ActG" << fx::match_gaussian("Xi", "Wgt")   | kwd::use("sigma", .3),;
```

3.3   Merging activities

The average is also the contextual merge. So we have two names for the same operation.

```
kwd::type("Ac1", "Map1D<Scalar>=100");
kwd::type("Ac2", "Map1D<Scalar>=100");
kwd::type("Ac3", "Map1D<Scalar>=100");
kwd::type("Ac",  "Map1D<Scalar>=100");
"Ac" << fx::average({"Ac1", "Ac2", "Ac3"});
"Ac" << fx::context_merge({"Ac1", "Ac2", "Ac3"});
```

A specific operation is allowed for merging contextual and external activity.

```
kwd::type("Ae", "Map1D<Scalar>=100");
kwd::type("Ag", "Map1D<Scalar>=100");
"Ag" << fx::merge("Ae", "Ac"); | kwd::use("beta", .5),
```

The formula is

$$\forall i, \ a_i^g = \sqrt{a_i^e(\beta a_i^e + (1 - \beta)a_i^c)}$$

3.4 Learning

Learning concerns the weights w_i of a map, once a best matching unit (BMU) π is defined. Learning consists of moving the weights to the current input ξ, in the surrounding of the best matching unit. The neighbooring function is $\mu(i, \pi)$, using Gaussian and triangular matchings μ_G or $\mu_\triangle$. The learning rule is:

$$\forall i, \ w_i^{t+1} = (1 - \alpha h)w_i^t + \alpha h \xi, \ \text{with } h = \mu(i, \pi)$$

```
kwd::type("Xi",   "Pos2D"            );
kwd::type("BMU",  "Pos1D"            );
kwd::type("Wgt",  "Map1D<Pos2D>=100");

"Wgt" << fx::learn_gaussian("Xi", kwd::prev("Wgt"),
                            "BMU") | kwd::use("alpha", .05), kwd::use("r",     .3);
"Wgt" << fx::learn_triangle("Xi", kwd::prev("Wgt"),
                            "BMU") | kwd::use("alpha", .05), kwd::use("sigma", .3);
```

3.5 BMU computation

A scalar activity distribution over a map serves as a basis for computing the BMU. The basic operation is the argmax, i.e. the position on the map where the activity is maximal. In case of ex-aequos, a random choice can be performed (see the "random-bmu" parameter). The activity distribution can be convoluated by a gaussian kernel before the argmax computation. The variance σ of that kernel is expressed in units sich as the map side as a size of 1, whatever the number of units actually in the map.

```
kwd::type("Act",         "Map1D<Scalar>=100");
kwd::type("BMU_noconv", "Pos1D");
kwd::type("BMU_conv",    "Pos1D");

"Act"        << fx::random()
"BMU_noconv" << fx::argmax("Act")       | kwd::use("random-bmu", 1.0);
"BMU_conv"   << fx::conv_argmax("Act") | kwd::use("random-bmu", 0.0),
                                         kwd::use("sigma", .05);
```

The BMU can also be updated from a previous BMU position, by modifying it toward the argmax (after convolution) with a step δ. This is used for relaxation processes. The number of convergence steps can be measured thanks to the "converge" operator.

```
kwd::type("Act", "Map1D<Scalar>=100");
kwd::type("BMU", "Pos1D");
kwd::type("NbSteps", "Scalar");

"Act" << fx::random()
"BMU" << fx::toward_conv_argmax("Act",
                                kwd::prev("BMU")) | kwd::use("sigma", .05),
                                                    kwd::use("delta", .05);
"NBSteps" << fx::converge({kwd::data("BMU")});
```

4 Implementation

4.1 Variables

Variables are stored in `.var` files. They are binary files. When unsigned integers are mentionned in the following, they are 8-byte unsigned values, starting from most significant bytes first (big endian). The file is organized as follows, in that order:

- 64 bytes: They contain an ascii version of the type, ended by '\n', and complemented with 0 bytes (padding) until the whole description's length is exactly 64-bytes.

- 8 bytes: An unsigned integer representing the cache size. This is used by the simulator to determine the size of the cache to be associated to that variable during the simulation.

- 8 bytes: An unsigned integer representing the buffer size s. Indeed, the file stores an history of the variable, i.e. the values from $[T, X, 0]$ to $[T, X, t]$. Although this theoretically represents $t + 1$ values, all of them may not be present in the file. The file is rather a circular buffer, with a limitted size s. So only values from $[T, X, t - s + 1]$ to $[T, X, t]$ are stored. Once determined in the file, the value of the buffer size cannot be changed.

- 8 bytes: The highest time in the file. This is the value t mentionned above. If the file is empty (i.e. even the the first $[T, X, 0]$ is not stored yet), the 8 bytes are 0xFFFFFFFFFFFFFFFF.

- 8 bytes: The next free position in the file. The file contain a range of s values, but it is a circular buffer. This bytes tells which index (starting from zero) is the next free position (i.e. where the $[T, X, t + 1]$ has to be stored.

- from 0 to at most $s \times (d + 1)$ bytes: This is the data. Each datum is a $[T, X, t']$ value, *preceeded* by a boolean byte. So if d is the number of bytes required for storing a datum, a slot requires $d + 1$ bytes. The file buffer contains then from 0 to at most s slots for storing values. If the boolean byte is 0 in a slot, the datum in that slot is considered as undetermined yet (i.e. busy). Otherwise, the d bytes following the boolean byte describe the datum value stored in this slot.