# SoMoX Review Guidelines

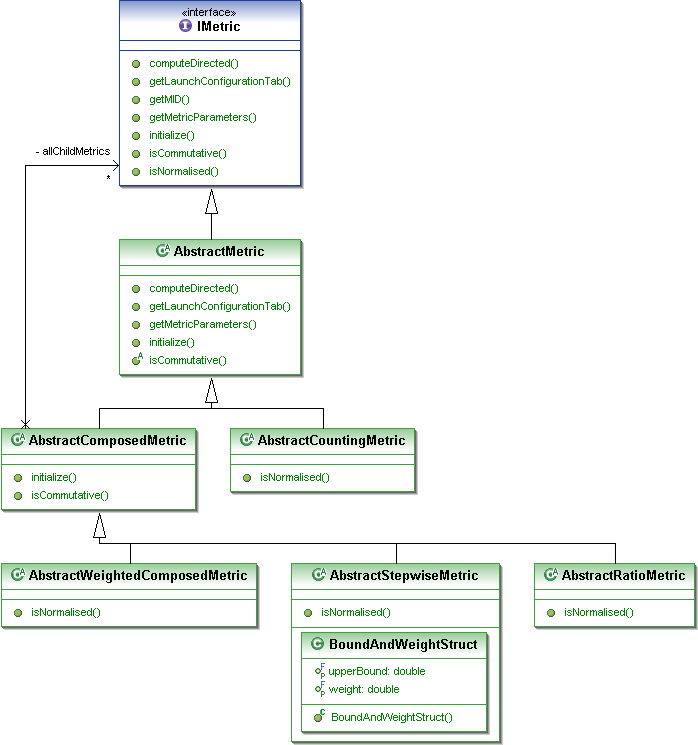
This document serves as starting point to understand or review the current version of the SoMoX code, especially of the plugin used to covert a SISSy model into a SAMM model.

Besides the plugins somox.core and somox.core.ui which are currently not fully used, SoMoX consists of the following interesting plugins:

|  |  |
| --- | --- |
| **org.somox.filter** | Filter framework used to filter any kind of collection, i.e., a list of classes |
| **org.somox.metrics** | Plugin containing all metrics used by SoMoX to cluster classes into components |
| **org.somox.metrics.dsl** | Plugin containing an xText DSL to describe new combined metrics (not yet fully implemented) |
| **org.somox.analyzer.sissymodelanalyser** | Plugin containing the clustering of classes into SAMM components |
| **org.somox.analyzer.sissymodelanalyser.ui** | Plugin containing the launch config used to start SoMoX in the current implementation, currently works around the core plugin |
| **org.jgrapht**  **uk.ac.shef.dcs.simmetrics** | Helper libraries for graph structures and string resemblance algorithms |

The main focus of the review should go to the metric computation and the model analyser.

## The metrics plugin

The metrics form a class hierarchy shown in the following figure:

Every metric implements the interface IMetric that provides all common functionality of a metric, especially the method computeDirected which computes a metric between 2 component candidates. It computes the value directed, i.e., the metric value for A 🡪 B is different from the value B 🡪 A. They are the same if the metric returns true for isCommutative.

There is a default implementation of the IMetric interface as an AbstractMetric which gives reasonable defaults for the methods it can implement. Based on it, there is a counting metric (a base metric) which counts for example the number of elements in a collection and a composed metric which derives its value by some computation using two or more other metrics. Special types of a composed metric are i) weighted metric which compute a weighted sum, ii) stepwise metric which scales another metric based on its value, and iii) ratio metric which computes the ratio of two other metrics.

All metrics used in SoMoX must be registered using a SoMoX metric extension point. They can be of two types: Basic metrics are directly implemented using the IMetric type hierarchy and metricDSLs use a metric DSL to describe especially new composed metrics.

## The analyser plugin

The analyser plugin uses a GAST model as input and by metric computation and clustering derives a component structure and a source code decorator from the code.

The analysis process starts in SimpleModelAnalyser and consists of two steps: First, identify a set of primitive components in the GAST model. Currently it uses the simple strategy to declare all top level classes and their nested classes as initial primitive components. If the class has an interface, it is used as component interface. If not, all public methods are used as derived provided interface. After that step, SoMoX enters the clustering step which assembles primitive components based on their inter-component relationships. This is encapsulated in a strategy and implemented in ComponentDetectionByClustering. The class first initialises all metrics registered for SoMoX via extensions. Then it computes a helper graph, in which the nodes contain all classes of all primitive components and the directed edges hold the results of all computed metrics for the adjacent nodes (e.g. counts of the number of accesses between the adjacent nodes; see getAccessGraph method and the following example figure).



Now the clustering itself starts. It consists of three steps in each iteration.

1. Compute relationship graph
2. Project graph to current threshold
3. Assemble new composite components from existing components

The first action computes a directed graph which has the current component candidates as nodes. The edges contain the lazily computed metric values for all metrics used on the respective edge starting from the top-level metric. The edges of the graph are computed in an optimized way. First, we examine the graph of the last iteration and determine what changed. For all connections between old nodes and new nodes and for all new nodes with each other, the edges are computed. Again, each single computation is optimized and uses a master-worker concurrency pattern. A unit of work is the computation of a pair of edges between two nodes (we compute pair wise and therefore use the isCommutative flag of a metric to avoid some computations). First, all work that needs to be done is collected and stored. Then all the work is presented to the workers who take one piece of work, compute it and return the results. If all work is done, the new graph edges are constructed.

In the second step the graph’s edges are filtered in a subgraph by removing all edges whose overall metric is not exceeding the current value of the threshold which indicated the assembly of a composite component . In the following figure, we see such a graph, where all edges are removed with DefaultCompositionIndicatingMetric <0.8.



The filtered graph has one weakly connected component which is composed into a composite component in the next step: For the third step we search in this filtered graph all weakly connected components (using the JGraphT API) and for each weakly connected component in the graph, we create a composite component. This is done by using the edges in the weakly connected component and trying to use them as assembly connectors (which usually makes sense for high coupling values). Intuitively, the components represented by adjacent nodes of a edge passing the threshold will be transitively assembled into a composite component.

For the example above the next graph clustering step produces the following graph where the newly created composite component is highlighted yellow.



Whenever we find components, we create new composite components and repeat then the steps 1.-3. If we do not find any further components, we decrement the threshold by the decrement step and enter the loop again until we reach the minimum threshold.

As a summary, the code aims at being clear and readable while still employing some performance tweaks like the master-worker pattern, the graph repair step, or the access cache.