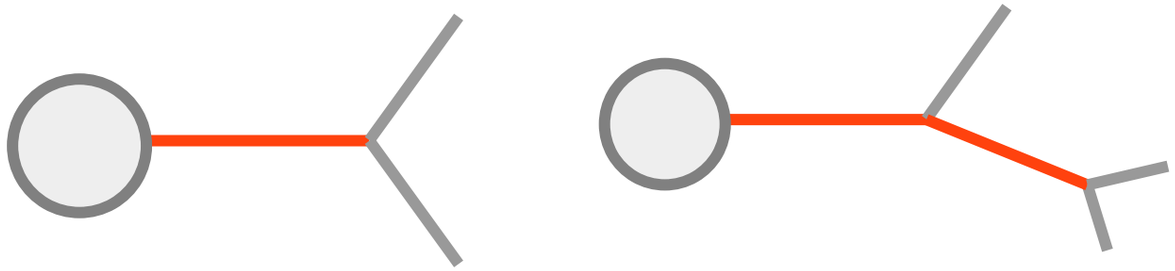
SModelS 3.0

Generalized Topologies With Graphs

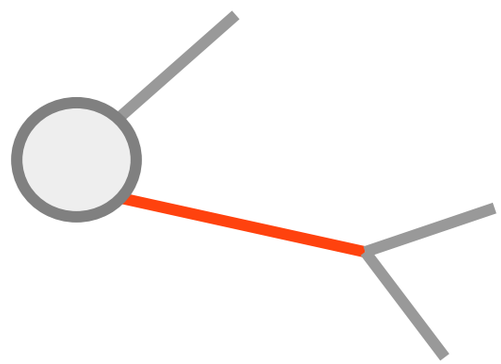
André Lessa

Goal

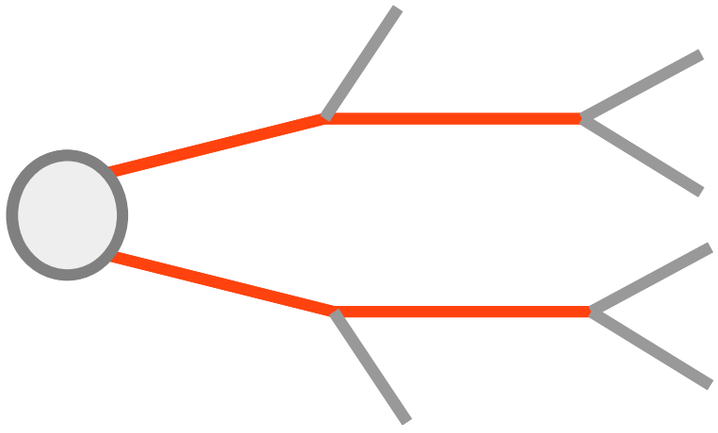
- Describe signal topologies with arbitrary shapes, such as:
  - Ressonant production



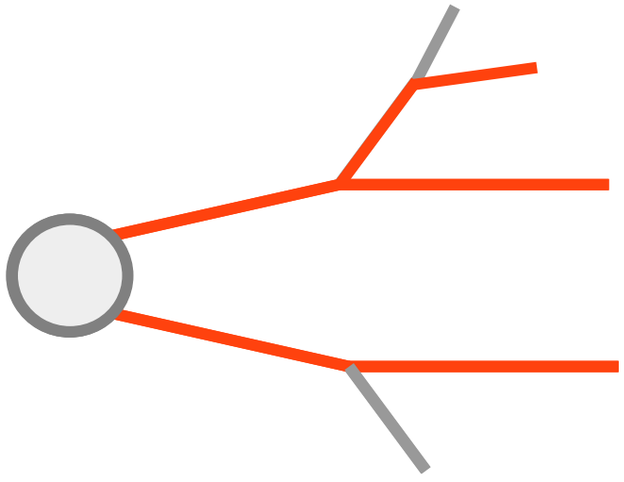
  - Associated Production



  - R-Parity Violating Decays



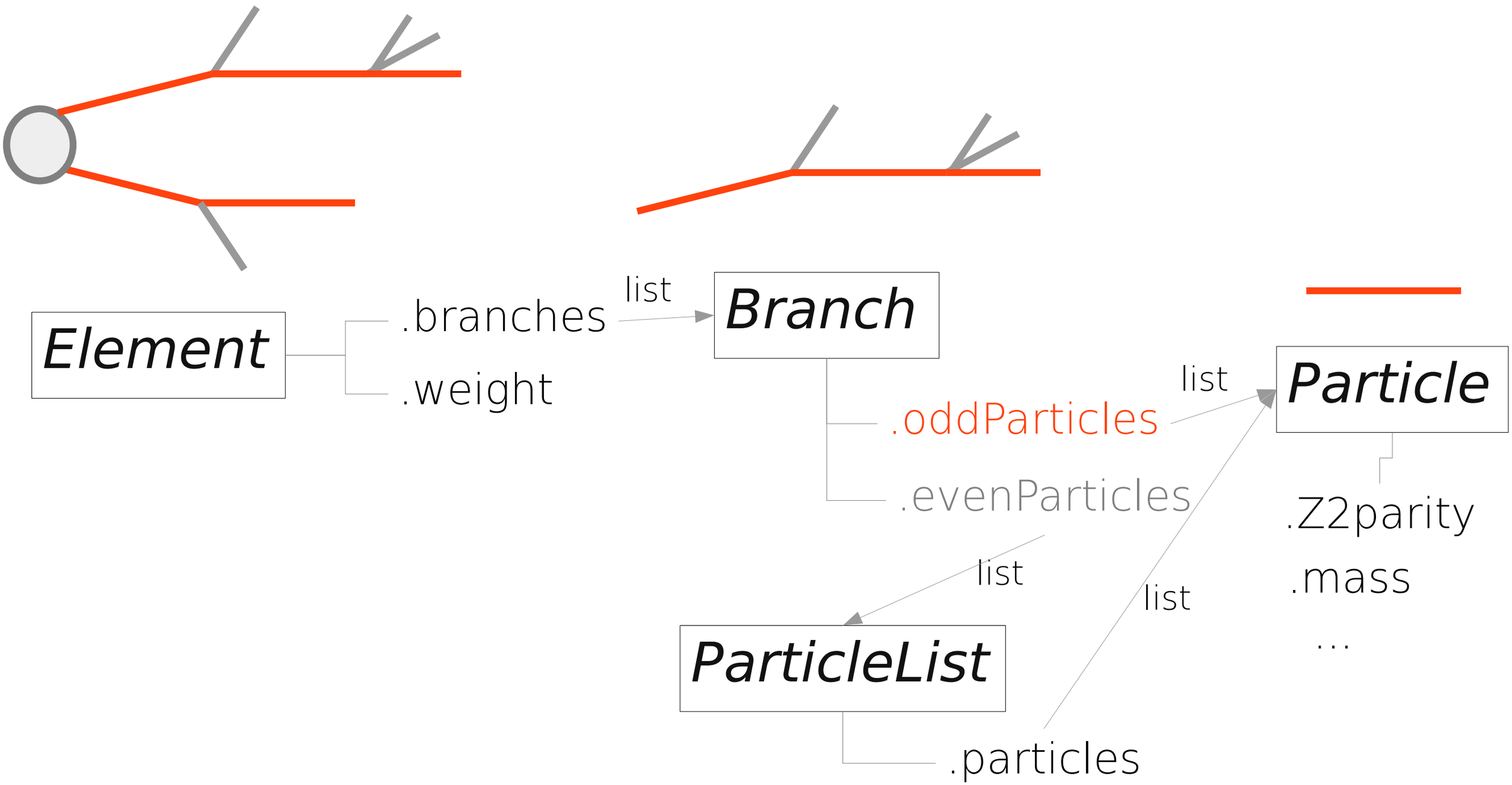
  - Non-Z_2 decays



...

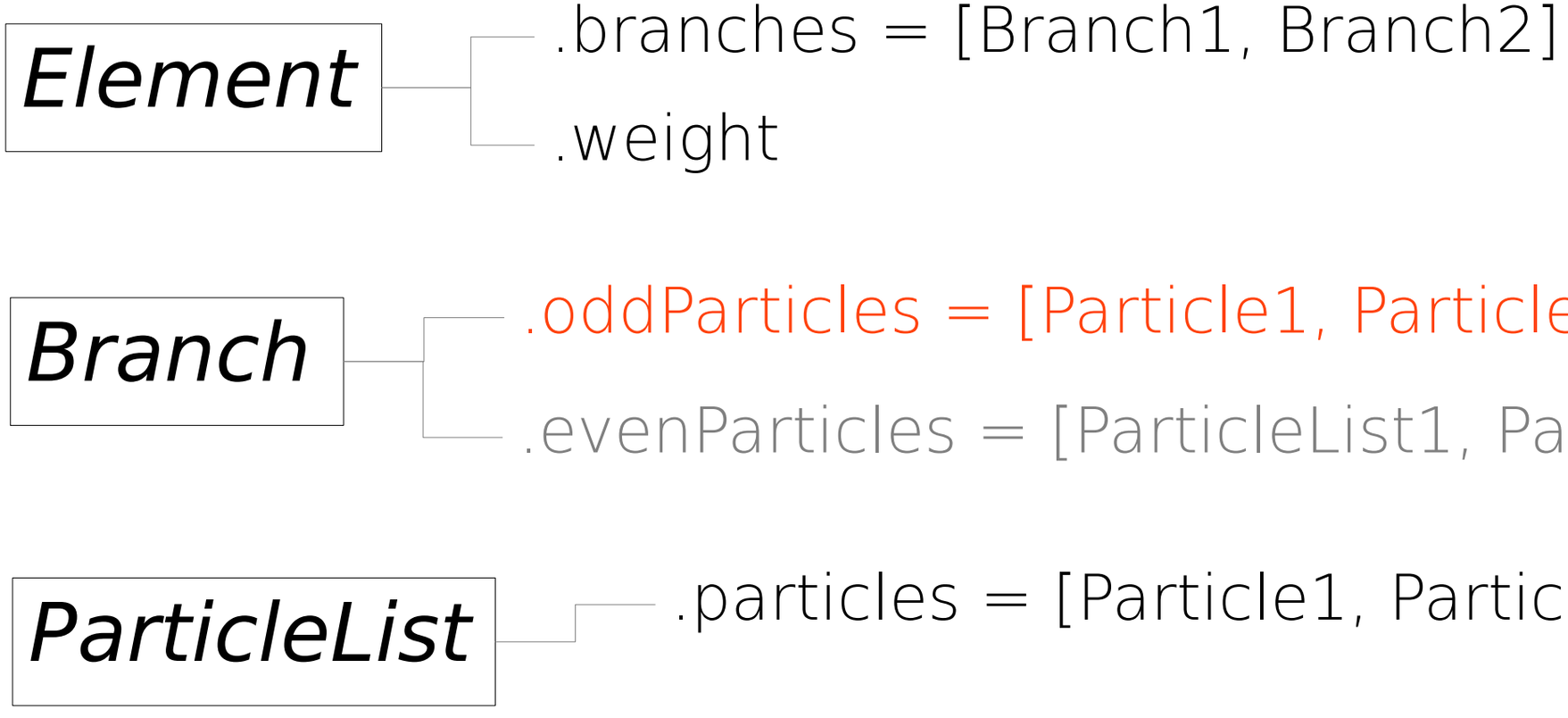
Current Status – SModelS 2.0

- Signal topologies (simplified models) are described by ***Elements:***



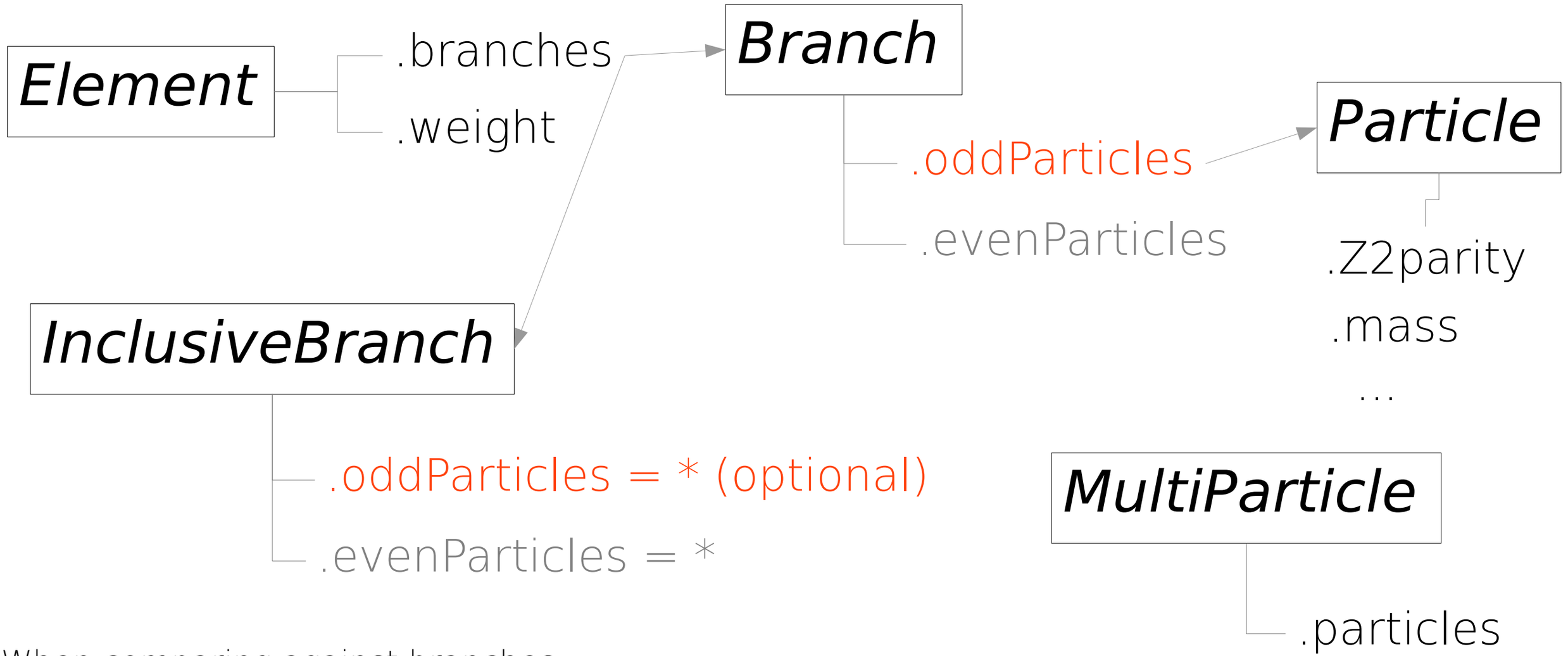
Current Status – SModelS 2.0

- Overall structure:



Current Status – SModelS 2.0

- Inclusive objects:



When comparing against branches:

- Checks if last BSM particle matches (if defined)
- Checks if intermediate BSM particles matches (if defined)
- Ignores evenParticles
- Ignores branch topology (number of vertices,...)

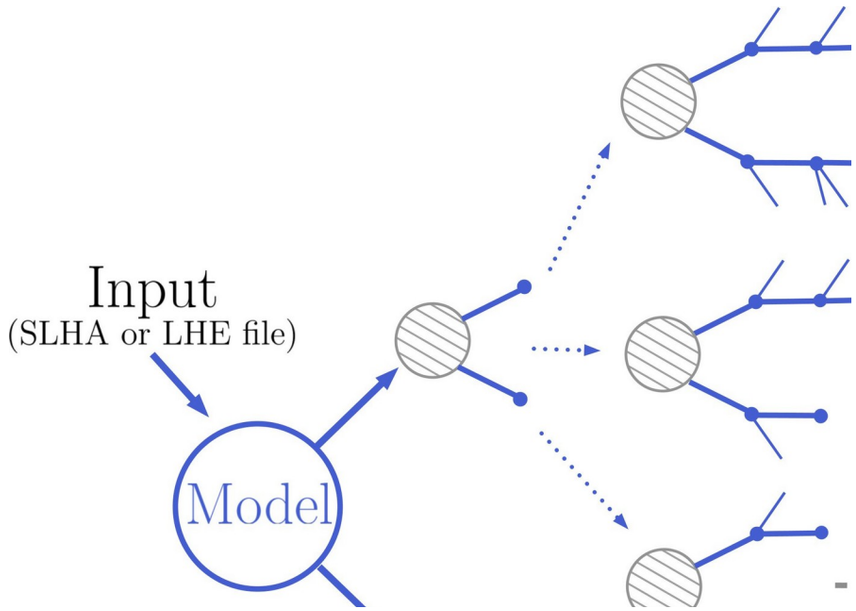
When comparing against particles:

- Checks if any particle in .particles matches

Current Status – SModelS 2.0

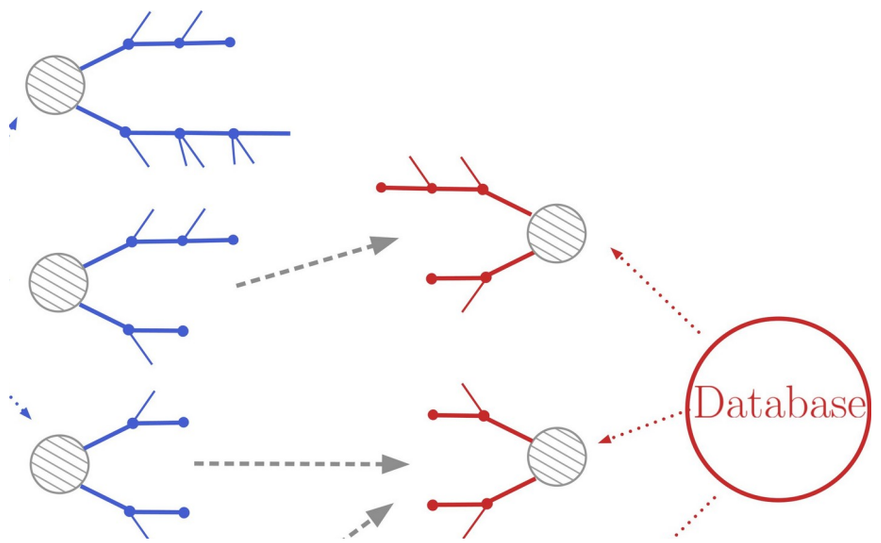
- Element comparison takes place when:
  - Decomposing the model

    (equal elements have their weights combined)



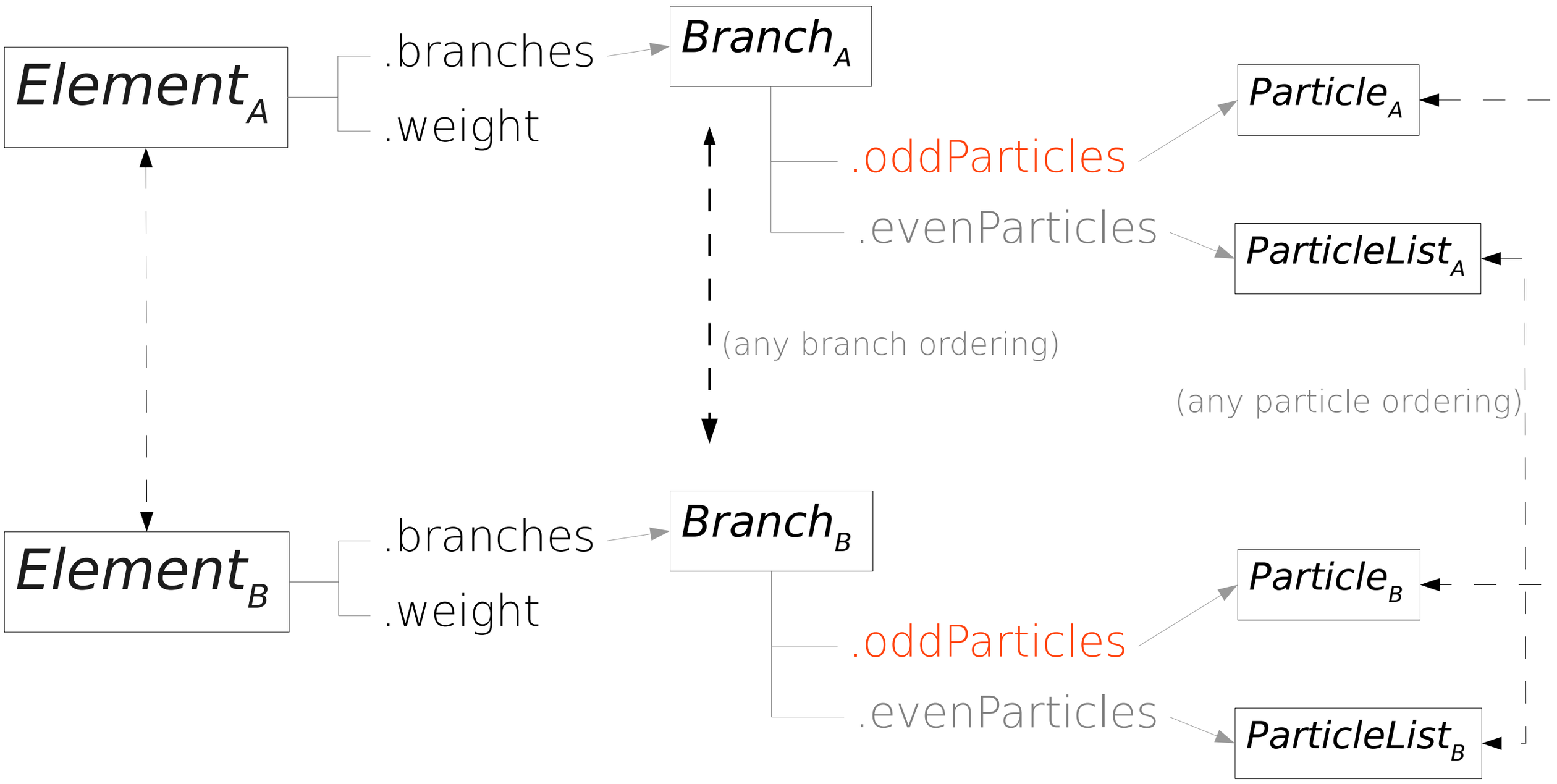
  - Matching against the database

    (uses inclusive objects)



Current Status – SModelS 2.0

- Element comparison: $Element_A = Element_B$?



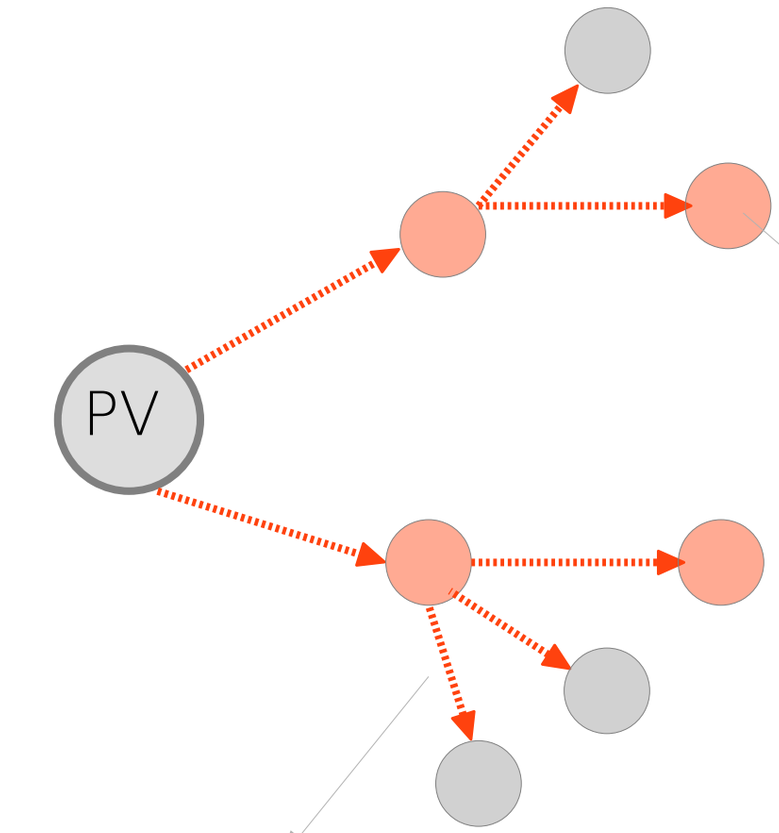
Current Status – SModelS 2.0

- A lot of the SmodelS structure is based on the 2-branch/Z_2 assumption

- However, it only affects the inner layers (Element class and its methods)

- How to move towards general topologies? → Graphs

Using Graphs to Describe Topologies



Nodes = Particles

Edges = "Decay"

Directed Graph
(Directed Rooted Tree)

- Can describe any topology
- Well stablished libraries (*networkx, ...*)

Implementation

- Main steps:
  - Text description ↔ graph ✓
  - Graph comparison ✓
  - Element compression ✓
  - Decomposition ✓
  - Inclusive objects ✓
  - Extracting relevant information from graphs (database) ✓
  - New database format (how to store data) ✓
  - Simplified missing topologies ✓
  - Printers ✓
  - …

Strings ⇔ Graphs

- We can use MadGraph-like language to convert the graphs to text and vice-versa:

  (PV → anyBSM(1), anyBSM(2)), (anyBSM(1) → e+, anyBSM(3), anyBSM(4)), (anyBSM (2) → e-, nu, anyBSM(5)) , (anyBSM(3) → nu,MET), ….

- We can use particle labels if we want to specify quantum numbers (as it is currently being done)

- We would like to keep some backward compatibility → convertion to/from bracket notation (for 2-branch/Z_2 elements)

Strings ↔ Graphs

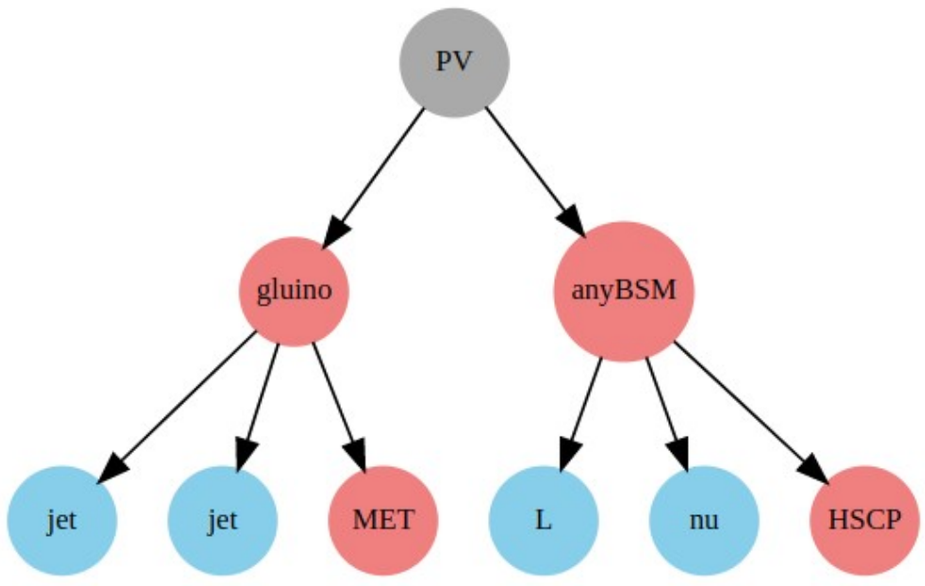
```
stringEl = "[ [ ['jet','jet'] ], [ ['L','nu'] ] ]"
output = bracketToProcessStr(stringEl,finalState=['MET','HSCP'],intermediateState=[['gluino'],['anyBSM']])
print(output)
```

```
(PV > gluino(1),anyBSM(2)),(gluino(1) > jet,jet,MET),(anyBSM(2) > L,nu,HSCP)
```

Convert process string to graph

```
procString = output
# Hack to create a theory element from a string:
expSMS = ExpSMS.from_string(output, model=finalStates)
tree = TheorySMS()
tree.add_nodes_from(expSMS.nodes)
tree.add_edges_from(expSMS.edgeIndices)

tree.draw()
print(tree.nodes)
print(tree.edges)
```




```
[PV, gluino, anyBSM, jet, jet, MET, L, nu, HSCP]
[(PV, gluino), (PV, anyBSM), (gluino, jet), (gluino, jet), (gluino, MET), (anyBSM, L), (anyBSM, nu), (anyBSM, HSC
P)]
```

SModelS 3.0 – Simplified Model Topologies

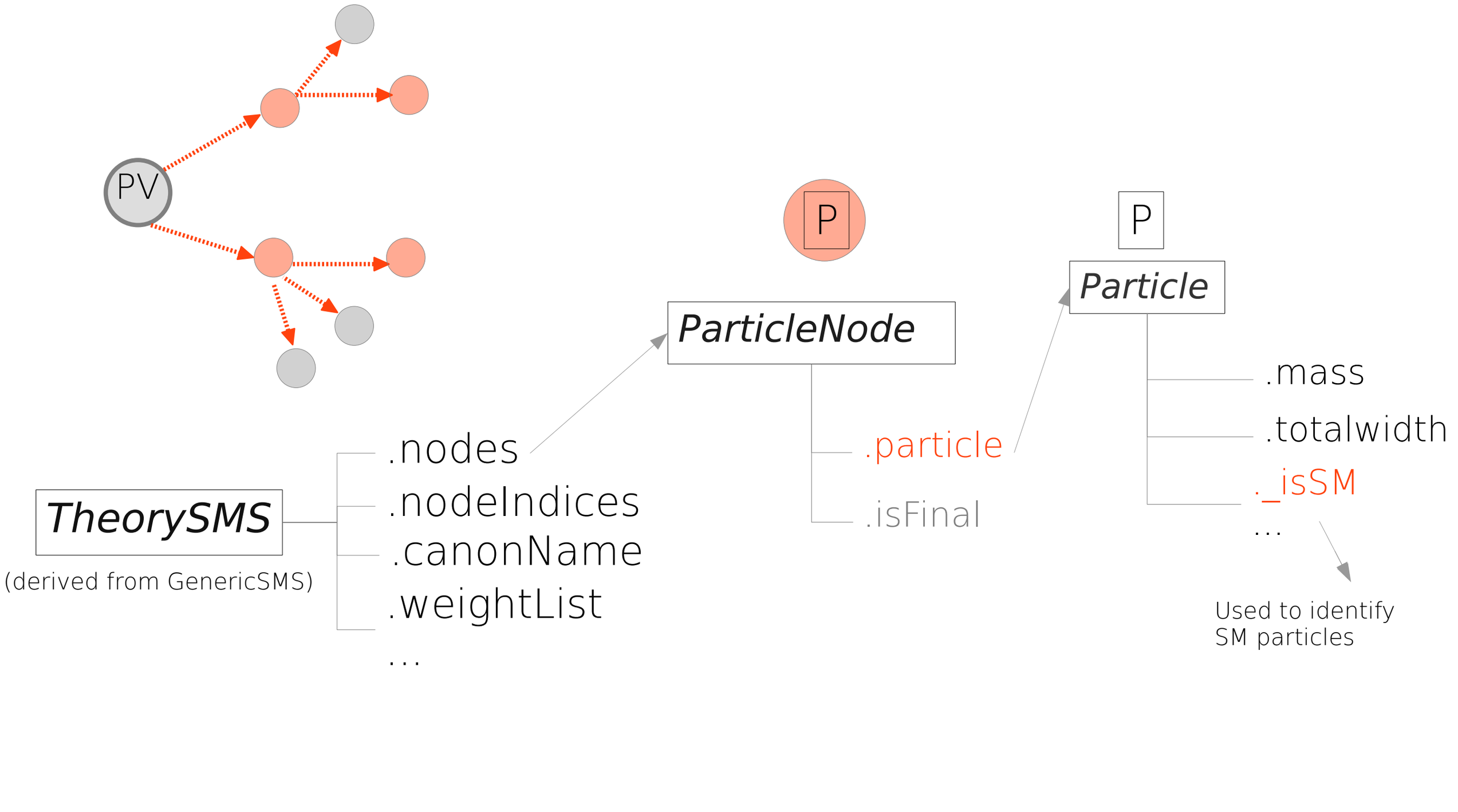
- In v3 “Elements” are renamed SMS (simplfied model spectra).
- Three types of SMS are defined:
  - GenericSMS
    - a base class implementing common features of graphs
  - TheorySMS
    - class derived from GenericSMS and is used to describe SMS from the BSM model (generated by decomposition)
  - ExpSMS
    - class derived from GenericSMS and used to describe database simplified models

SModelS 3.0 – Simplified Model Topologies

- The main differences between TheorySMS and ExpSMS are:
  - TheorySMS objects have a well defined comparison, so a sorted list can be built.
  - ExpSMS objects do not have a well defined comparison, since they can contain inclusive nodes.
  - Nonetheless a matching (equal/not equal) between ExpSMS and other SMS objects can be defined and has to allow for any node ordering.
  - Both TheorySMS and ExpSMS objects can have their structured sorted.

SModelS 3.0 - TheorySMS

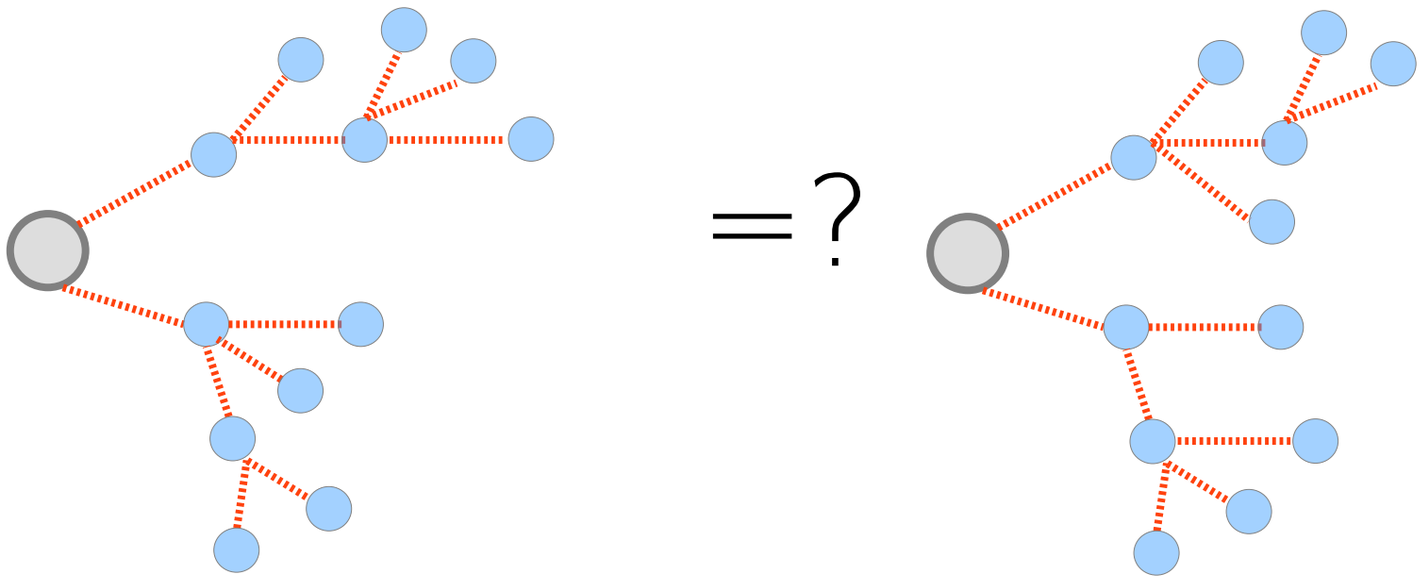
- Signal topologies from the BSM decomposition are described by Theory**SMS** topologies*:*



TheorySMS Comparison

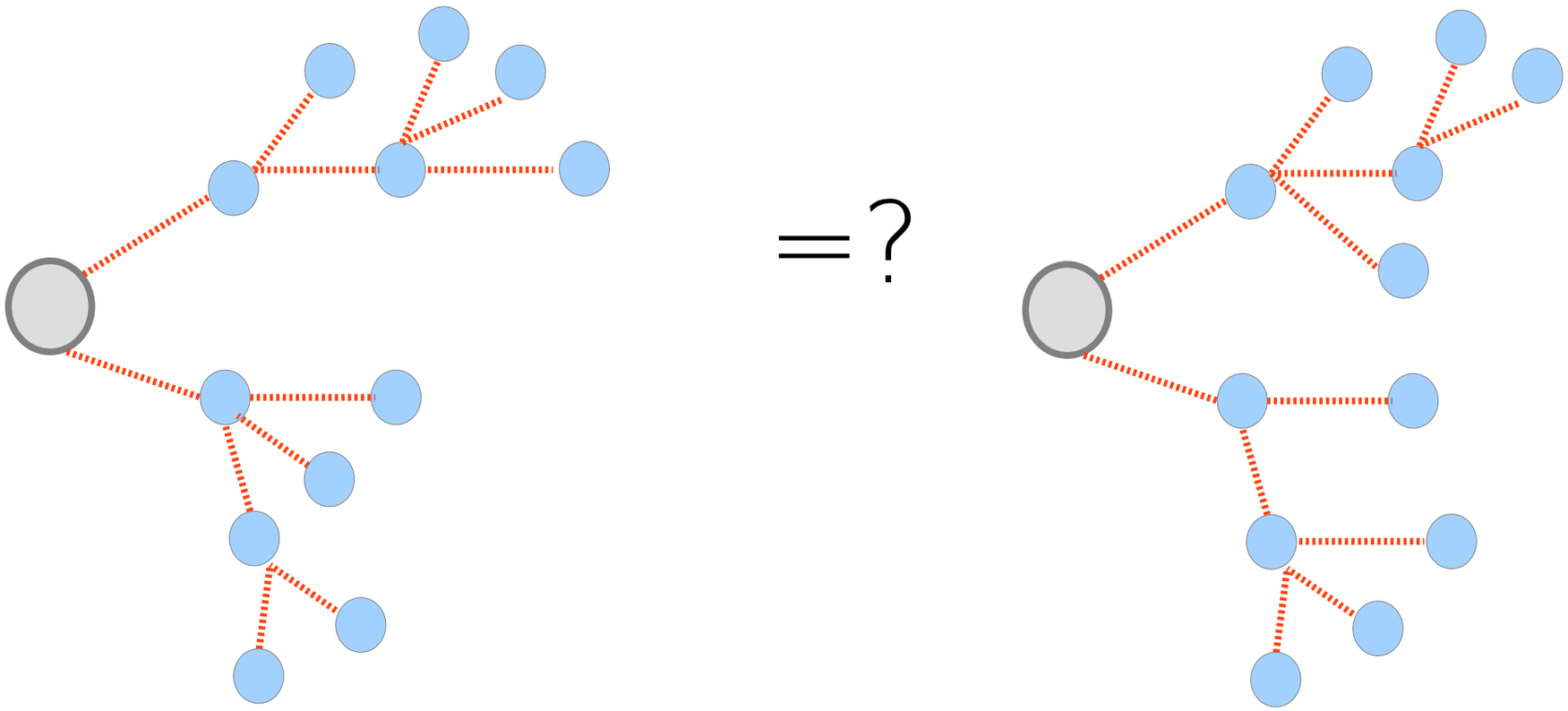
- Comparing SMS is no longer simple, since we have arbitrary shapes (topologies).
- But it can still be done in 2 steps:

1) Shape (topology) matching
2) Particle comparison



TheorySMS Comparison

1)Shape (topology) matching:



=?

- The above shapes are equivalent, since the ordering of the nodes in each decay is irrevalant.
- Easy using canonical labels (names)!

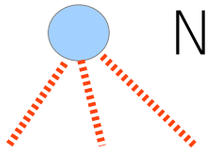
Canonical Name

- We define the canonical (topology) label as:

  i) For each final node (leaf):

  - Name → 10

  ii) For each decayed node:

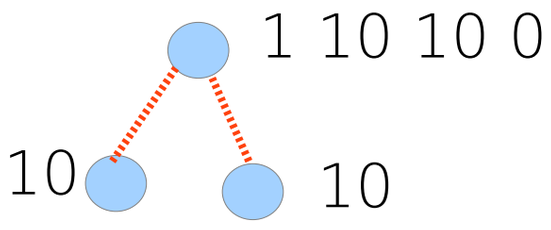
  

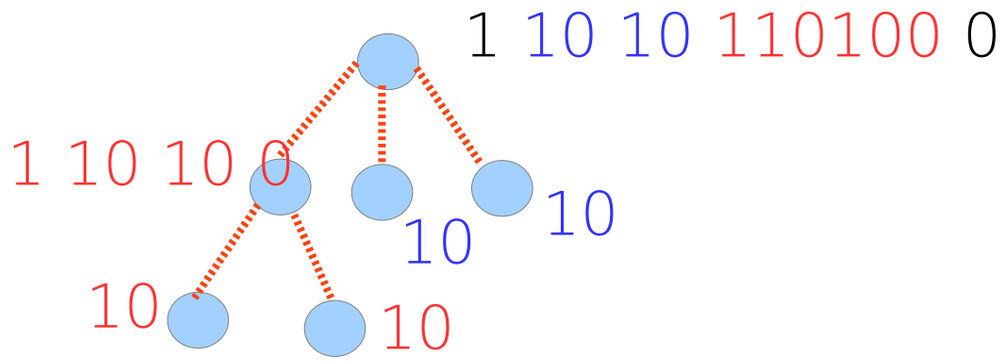
  Name → 1 (sorted daughters name) 0

  iii) Build name recursively → Graph name = PV name

Similar to a nested bracket notation! (1 → left bracket, 0 → right bracket)

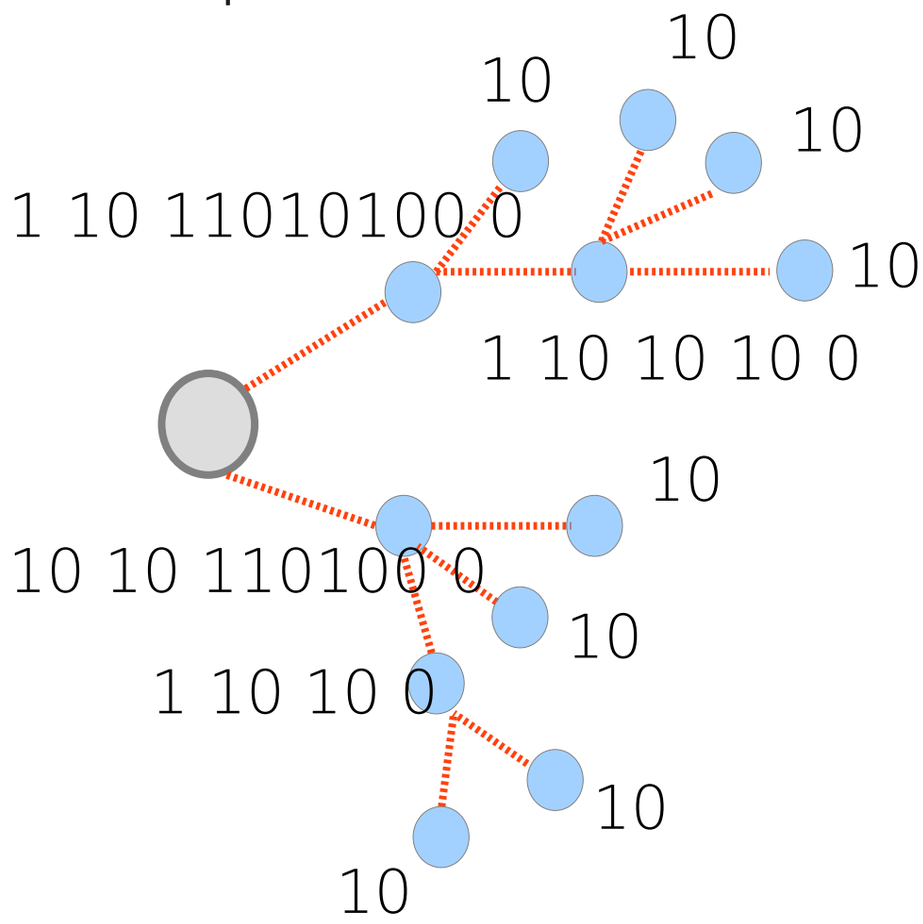
- Examples:



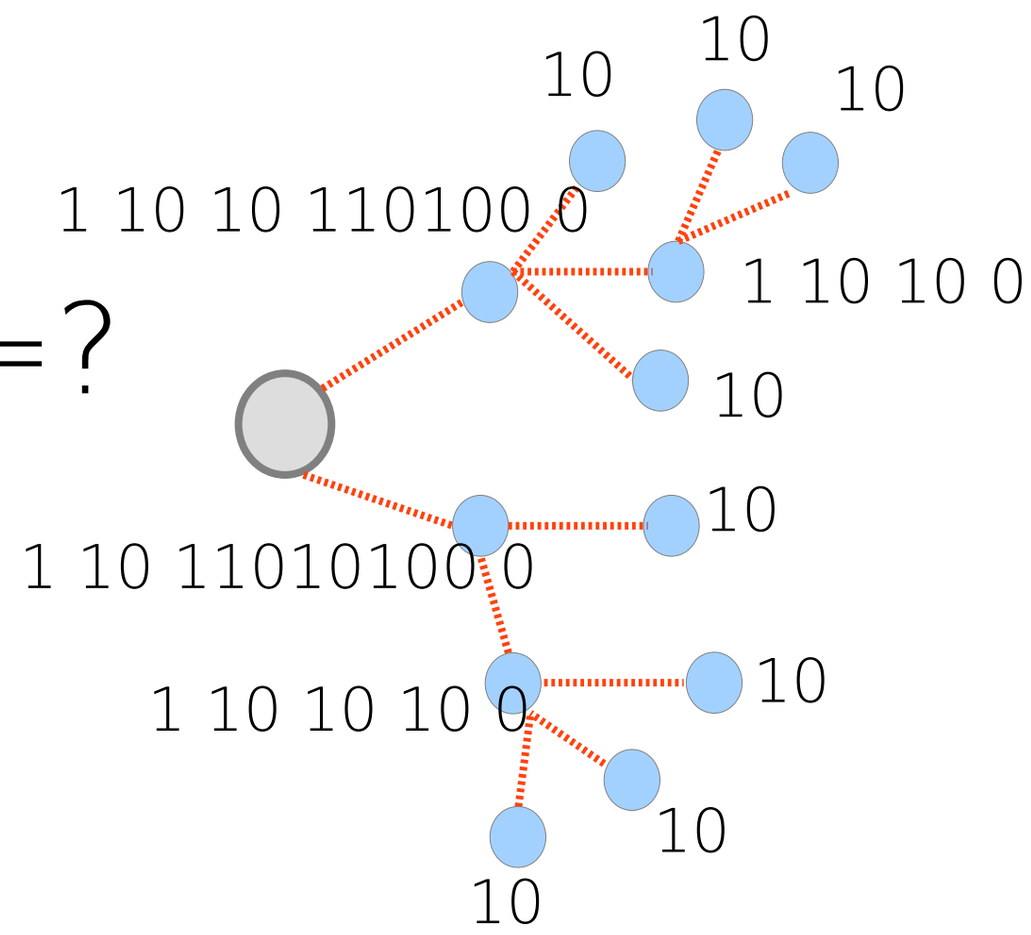


Canonical Name

- Example:



=?



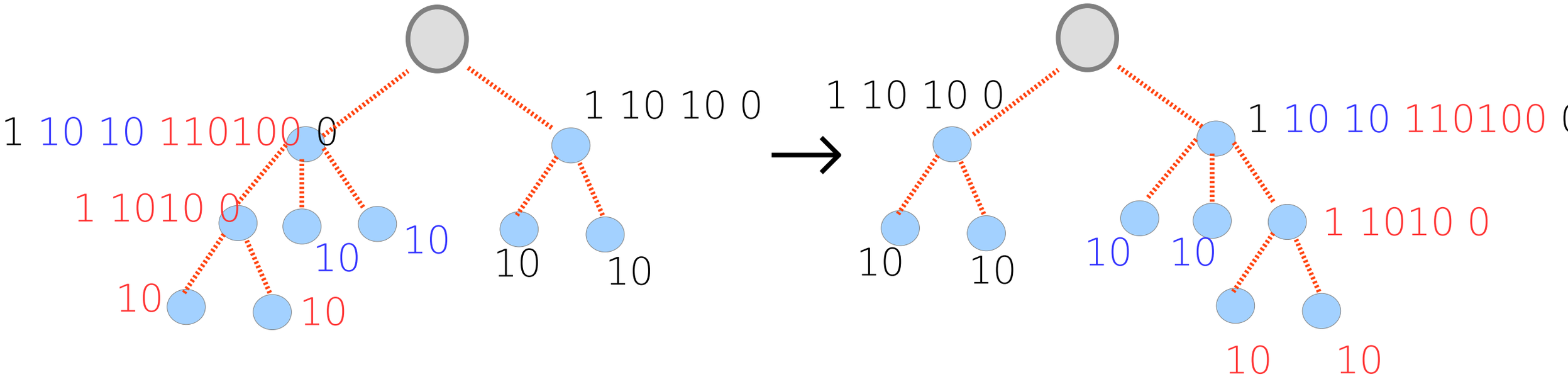
Name 1:
1 110101101000 110110101000 0

Name 2:
1 110101101000 110110101000 0

Name 1 = Name 2

Canonical Name

- The canonical name can also be used to sort the nodes from the same parent and structure the tree in a default format

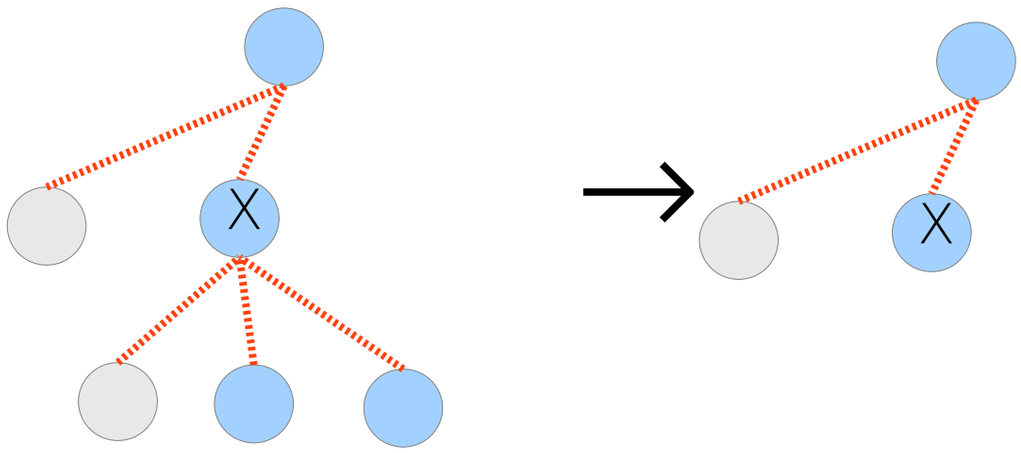


- For equivalent nodes (same name) we sort according to particle and then daughter nodes

Compressing TheorySMS

- How to define mass and invisible compression?
- It has to be applied node by node!

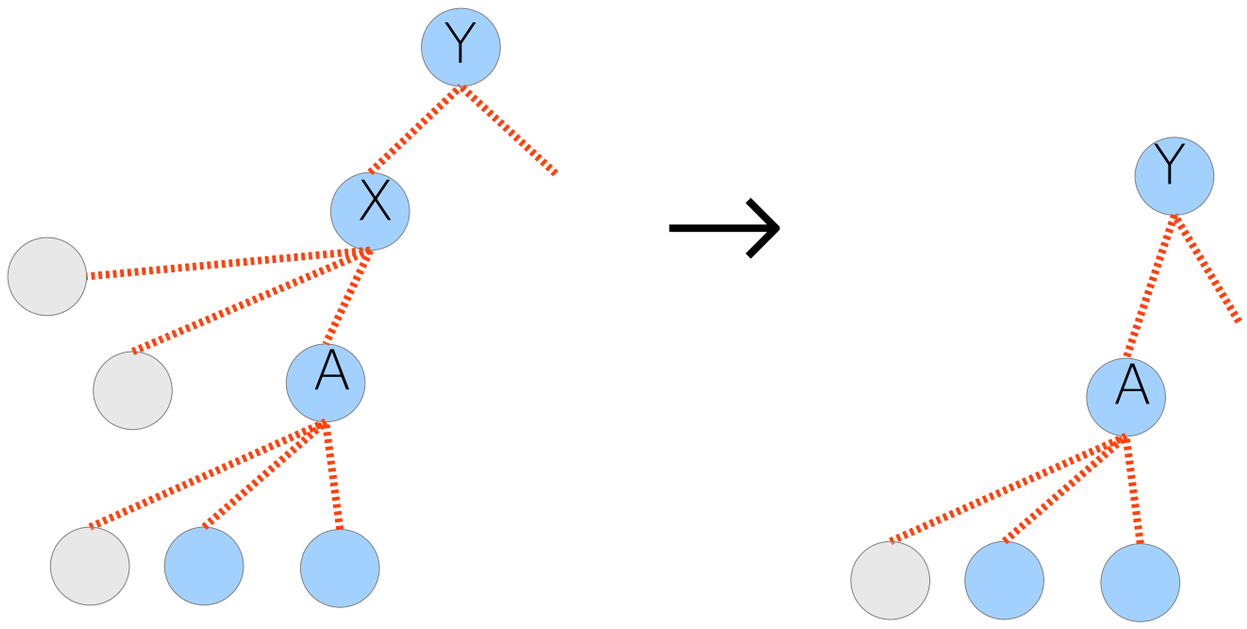
1) Invisible Compression



- Node X is compressed if:
  - All its daughters are invisible
  - All its daughters are “final states”
  - X decays promptly ***or*** is invisible

Compressing TheorySMS

2) Mass Compression



- Node X is compressed if:
  - decays promptly
  - decays to a ***single*** BSM particle
  - and

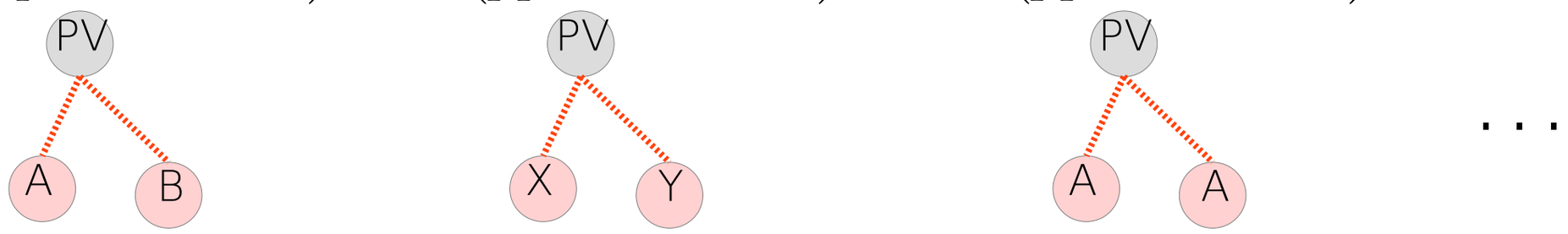
$$m_X - m_A < \Delta m$$

Decomposition

- Decomposition follows a similar algorithm, but generalized to deal with arbitrary graphs.

- First, starting with the model (production cross-sections and particles), create all simple SMS of the form:

$\sigma\left(pp \to A + B\right)$ $\sigma\left(pp \to X + Y\right)$ $\sigma\left(pp \to A + A\right)$

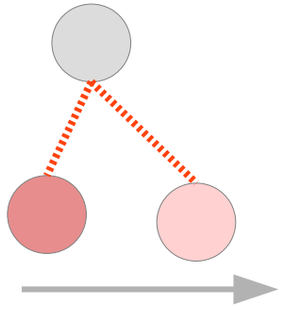


- Only SMS with cross-sections larger than sigmacut are kept

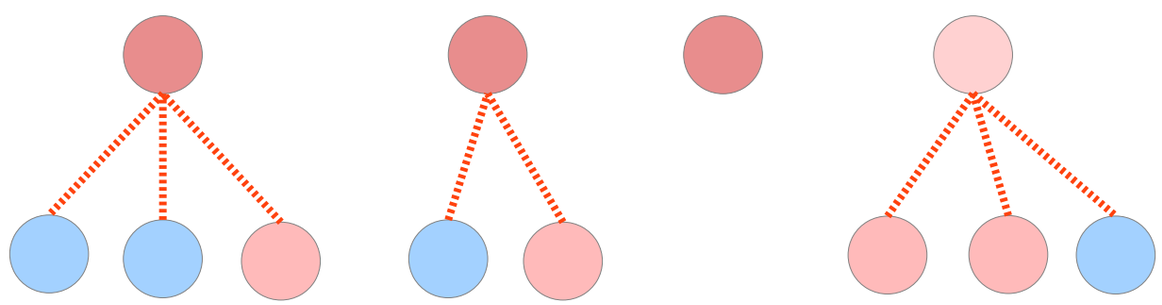
Decomposition

- For a given initial SMS:

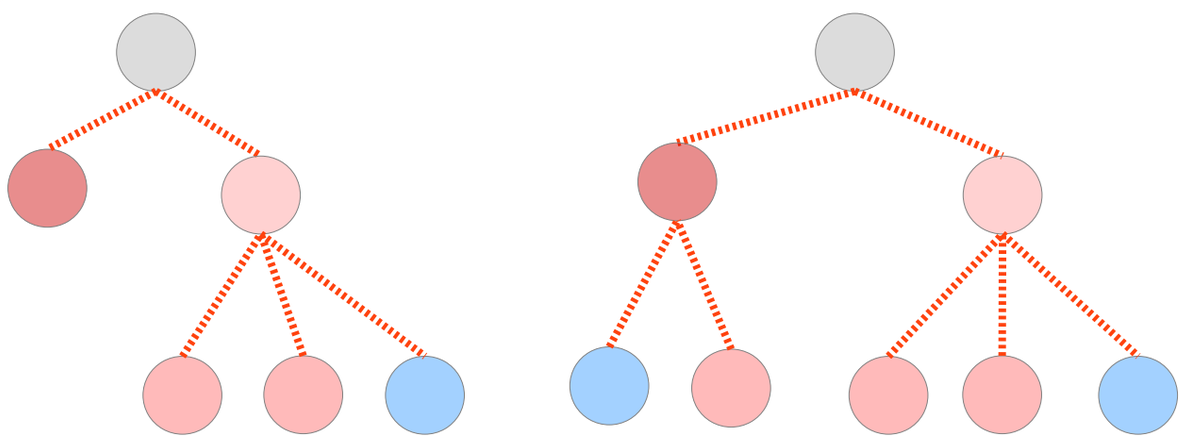
1) Iterate over the “final states”



2) For each final state construct all possible decays (including possible stable state)



3) Add all decays to original SMS

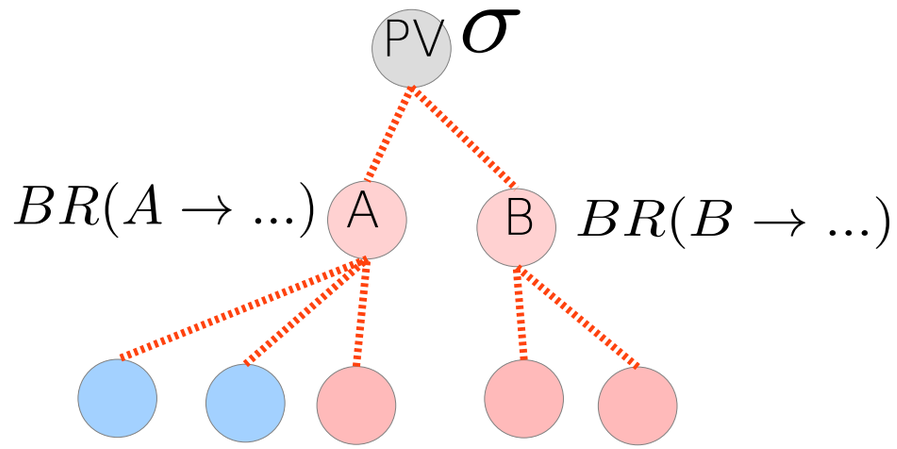


4) Apply step 1) to all SMS created by the last step

Decomposition

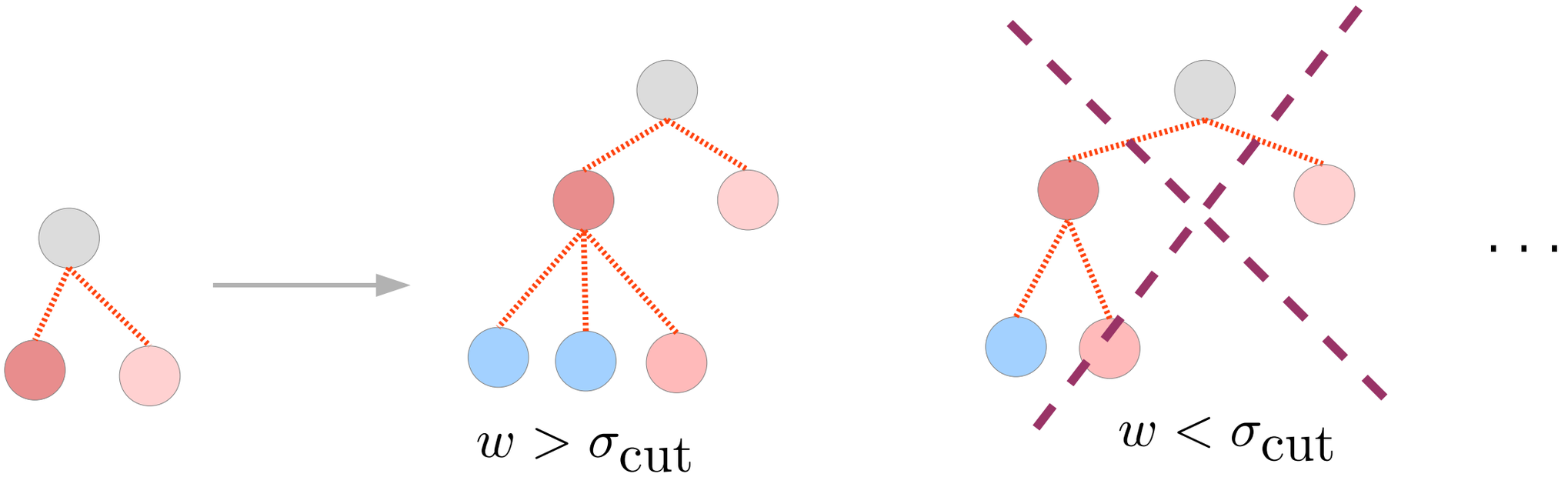
- Nodes can be tagged with the attribute ".isFinalState = True" to avoid further decays
- The SMS has the attributes:
  - prodXSec = list of production cross-sections (distinct sqrts) for PV
  - maxWeight = maximum weight (largest production cross-section times BRs)
  - decayBRs = total BR (product of BRs appearing in the SMS)

$$w = \sigma \times BR(A \to ...) \times BR(B \to ..)...$$



Decomposition

- During step 3) any SMS with maxWeight smaller than sigmacut is ignored



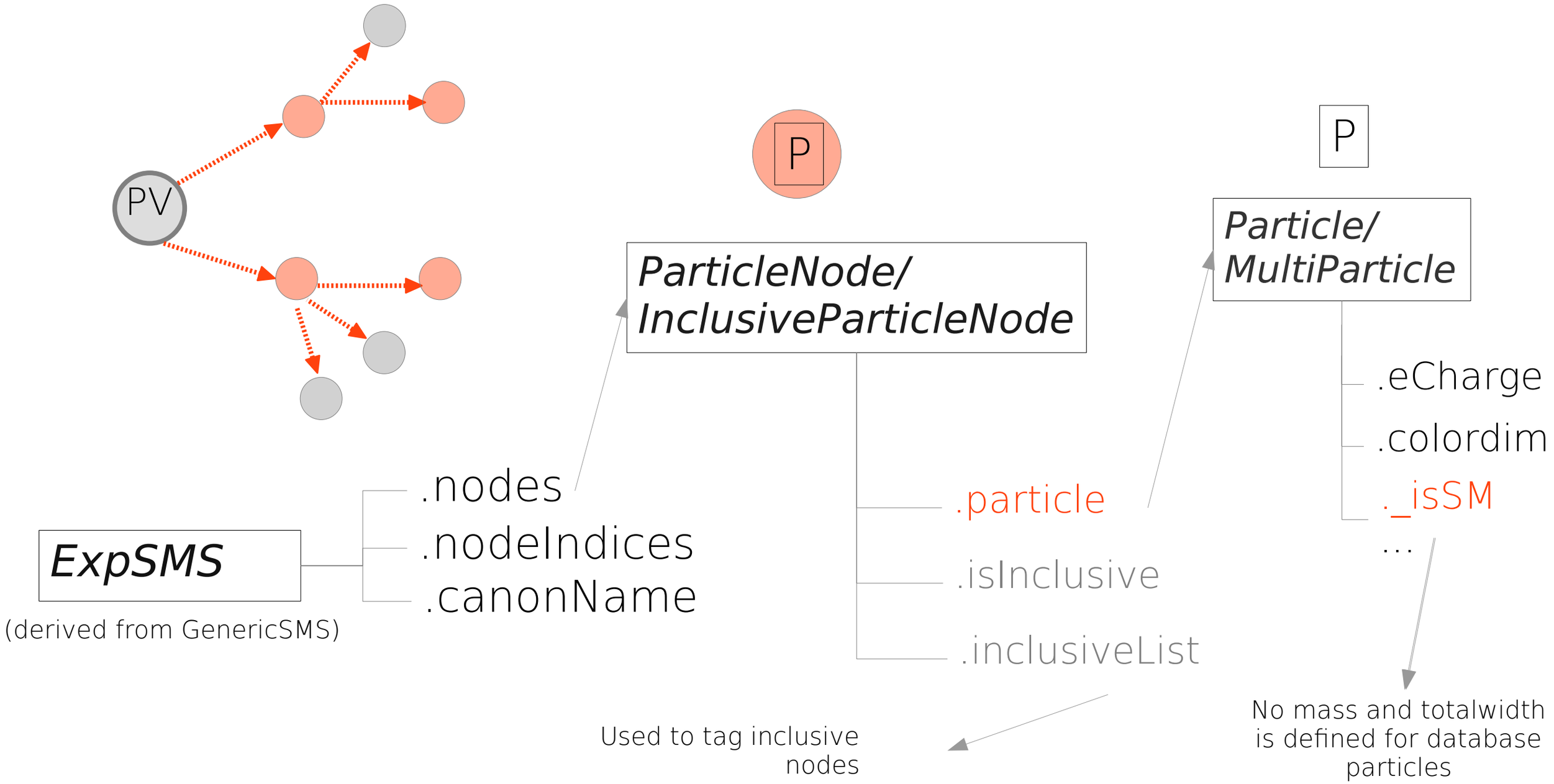
- In the end only SMS where all final state nodes have finalStates = True are kept (avoids intermediate undecayed trees generated by failing to pass sigmacut)

- There is a lot of duplicated steps in the decompositon, since everytime a particle appears all of its possible decay trees have to be generated again. For instance, pair production of the same particle will generate exactly the same decay branches twice. However, since the possible decays are filtered by sigmacut, it depends on which position the particle appears (how small is the weight leading to the particle). Therefore previously generated cascade decays (under some effective sigmacut assumption), can not be easily recycled.

→ A better solution still to be found?

SModelS 3.0 - ExpSMS

- Simplified model topologies used for defining experimental results are described by Exp**SMS** topologies*:*



SModelS 3.0 - ExpSMS

- Unlike Theory**SMS** topologies, Exp**SMS** topologies can hold inclusive nodes and inclusive particle definitions (multiparticles).

- Exp**SMS** can be generated from strings (as appearing in TxNames constraints and conditions)

- Experimental topologies can be **matched** to other (ExpSMS or TheorySMS) topologies.

- The matching has to take into account inclusive node and particle definitions and all possible node orderings.

- The matching is done in two steps:

    1) Shape (topology) matching (for ExpSMS containing inclusive nodes, the shape is not defined and will match any other shape)

    2) Particle matching (must take into account any node ordering and inclusive nodes)

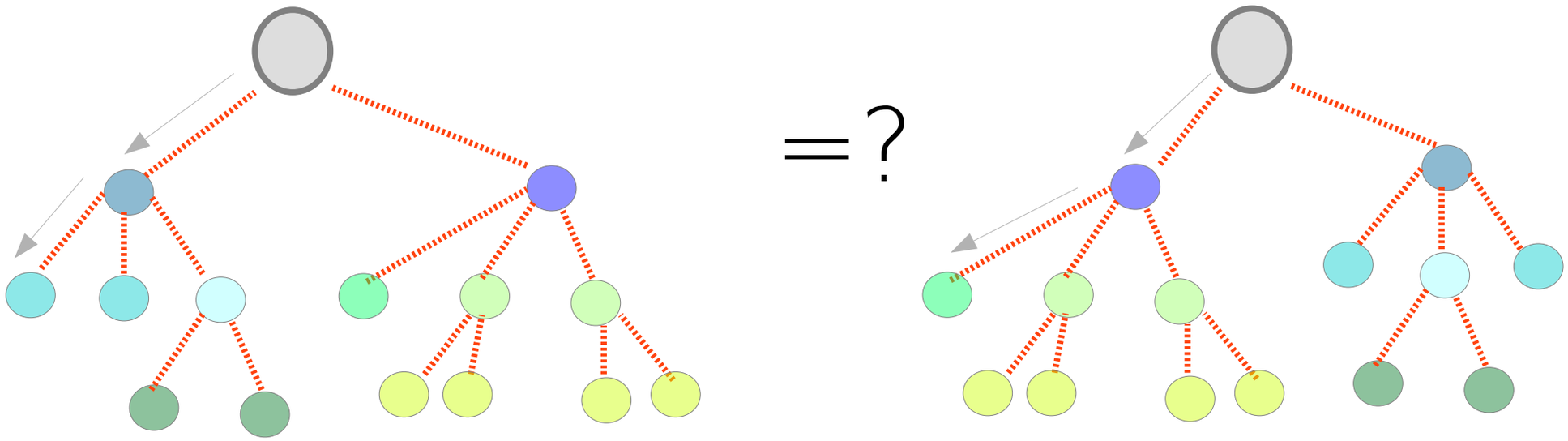
ExpSMS Matching

1) Shape (topology) matching:

- Simple comparison of canonical names
- If the ExpSMS contains an inclusive node, its canonical name is defined as “*” and will match any canonical name.

ExpSMS Matching

2)Particle/Node Matching

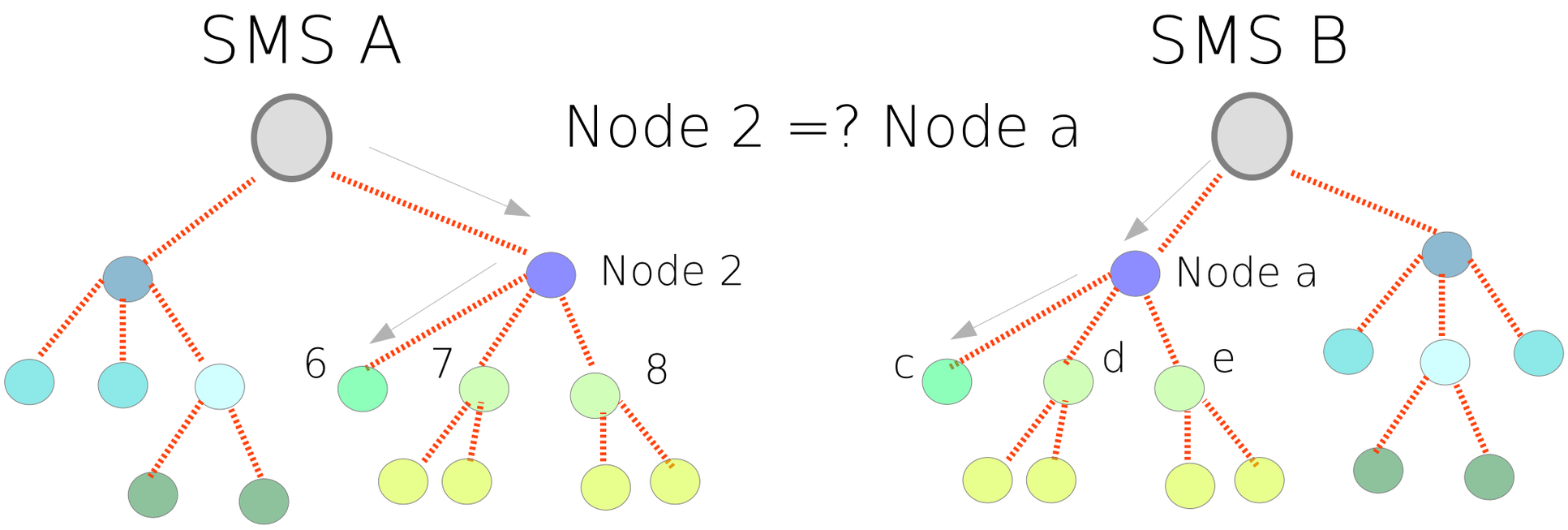


- Two nodes match if:
  I. Their particles and canonical names match
  II. Their daughter nodes match
- Two SMS match if their root nodes match, which means the SMS have matching canonical names and all nodes have matching nodes.

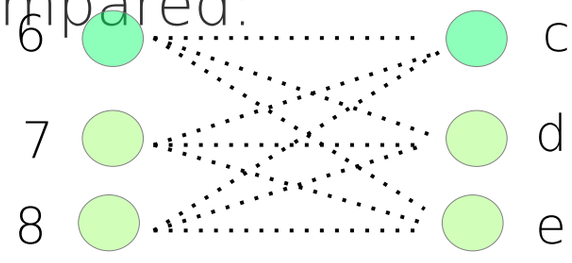
ExpSMS Matching

2)Particle/Node Matching

- When matching two sets of daughters from the nodes being compared, the following procedure is adopted:



I. All possible pairs of daughters are compared:

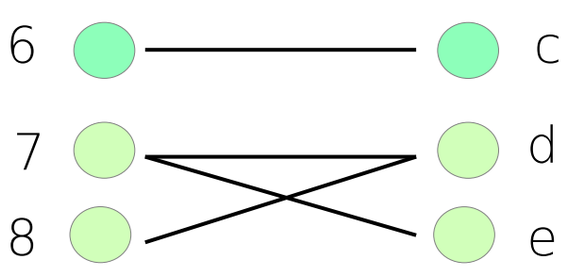


(to improve perfomance, it is only necessary to compare nodes with the same canonical name)

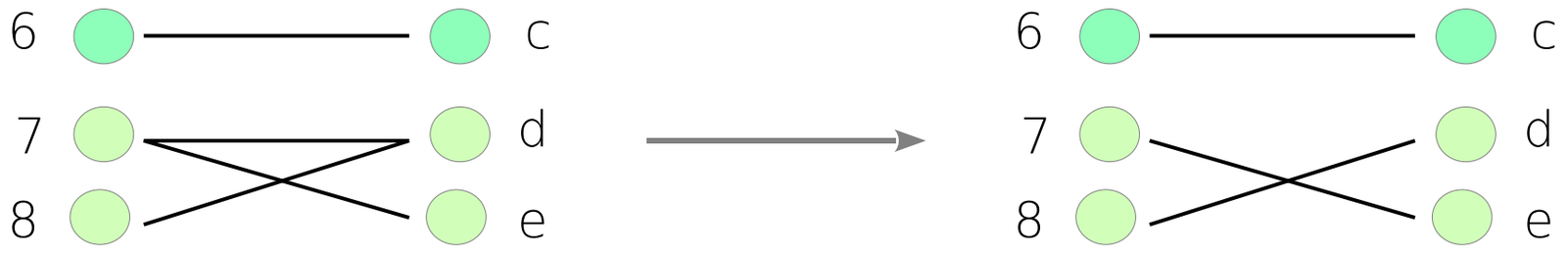
ExpSMS Matching

2)Particle/Node Matching

II. A bipartite graph is constructed where the left nodes are the daughters for SMS “A”, the right nodes the daughters from SMS “B” and the edges connect the pairs of daughter nodes which have matched.



III.A maximal matching algorithm is used to determine the maximal number of matchings (edges which do not share common nodes)

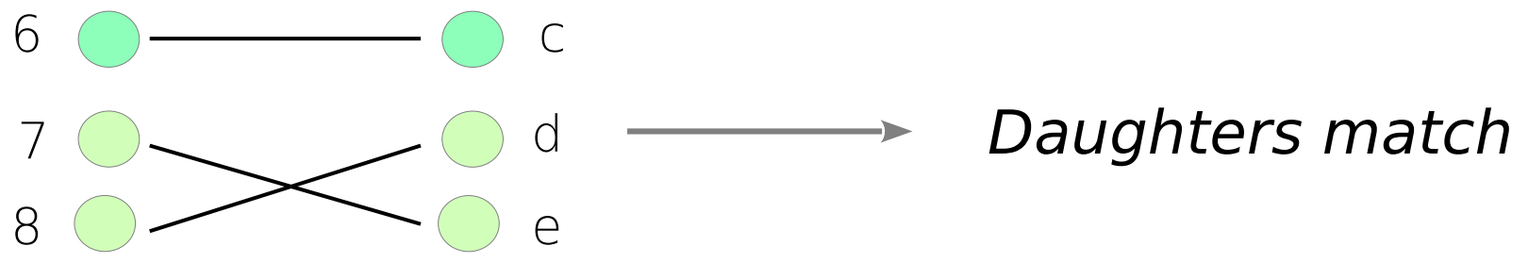


maximal matching = {6 : c, 7 : d, 8 : e}

ExpSMS Matching

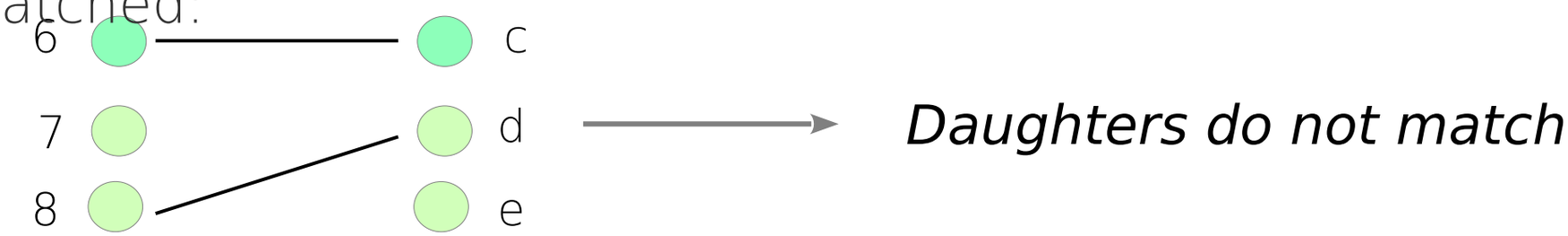
2)Particle/Node Matching

IV. If all left nodes and right nodes have had matches, the daughter nodes are considered as matched and the matching dictionary is kept



maximal matching

Otherwise the daughters are NOT matched:



maximal matching

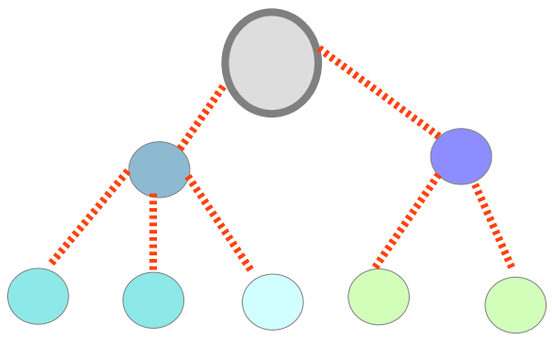
- With this procedure, the SMS matching checks for any possible node ordering.

ExpSMS Matching

- If more than one ordering matches we keep the “first one” (arbitrary)
- In v2 we already have this issue, but it is simpler:
  - only two possible branch orderings (we always check both)
  - in a given vertex we compare ParticleLists (SM particles) and cache the comparison result

- In order to “reproduce” the v2 results, in v3 we check the matching for 2 possible branching orderings (if the TheorySMS has a 2-branch structure):

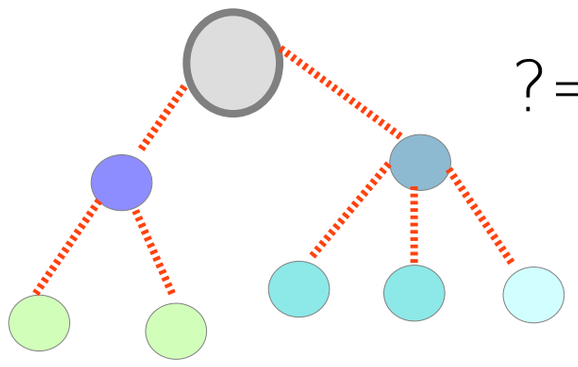
SMS A



?= SMS B

or

SMS A′



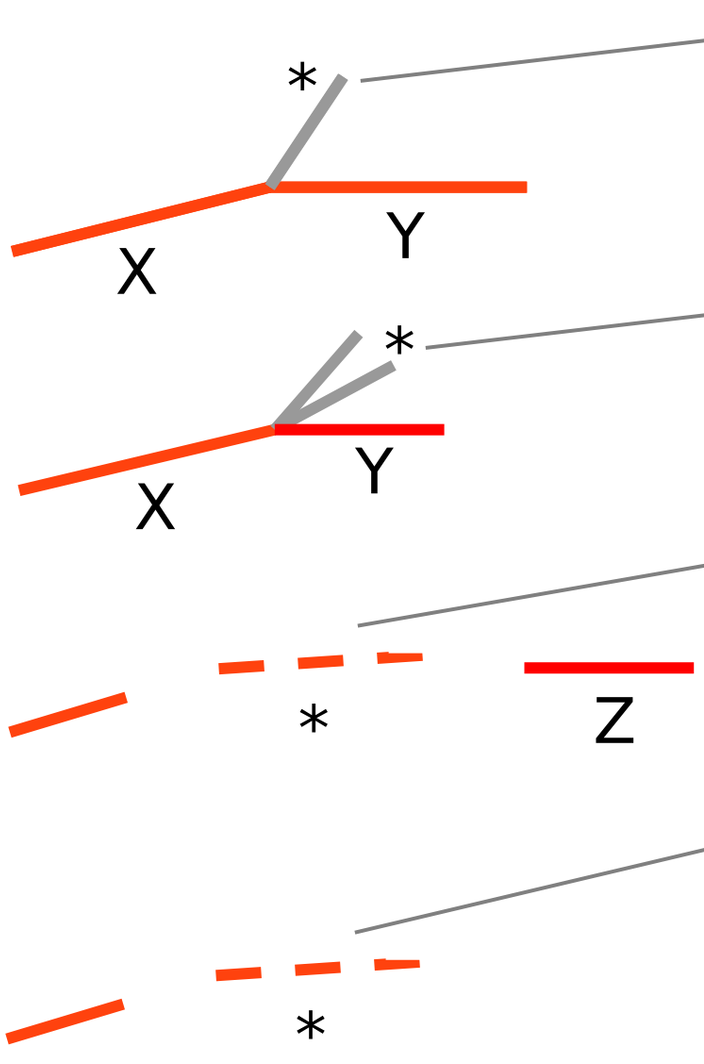
?= SMS B

ExpSMS Matching

- Note that if only one ordering matches SMS B, comparing both orderings will return the same result (since all possibilities are checked).
- However, if both match, we get two possible matching orderings. In this case:
  - In order to always select a unique ordering, the two orderings are sorted according to their respective dataPoints (which holds all the information required for interpolation of the TxName grid) and their reweighting factor.
  - This way a specific branch ordering is singled out.
  - The above result, however, is limited to the Z_2 case, not very efficient (since we need to do two very similar comparisons)
  - The general case would require to compute all possible matchings, sort them and select one. However this is technically much more challenging and might reduce the performance quite a bit.

Inclusive Objects – SModelS v2

- In SModelS v2 we make use of
  - *Inclusive Particles*: matches any particle
  - *Inclusive Branches*: matches other branches (may or may not include comparison of intermediate and final BSM states)



Any (single) SM final state:
(InclusiveParticle → [['*']])

Any arbitrary number of SM particles:
(InclusiveBranch→ ['*'] + intermediate and final states defined)

Arbitrary SM final states and BSM intermediate states:
(InclusiveBranch→ ['*'] + final states defined)

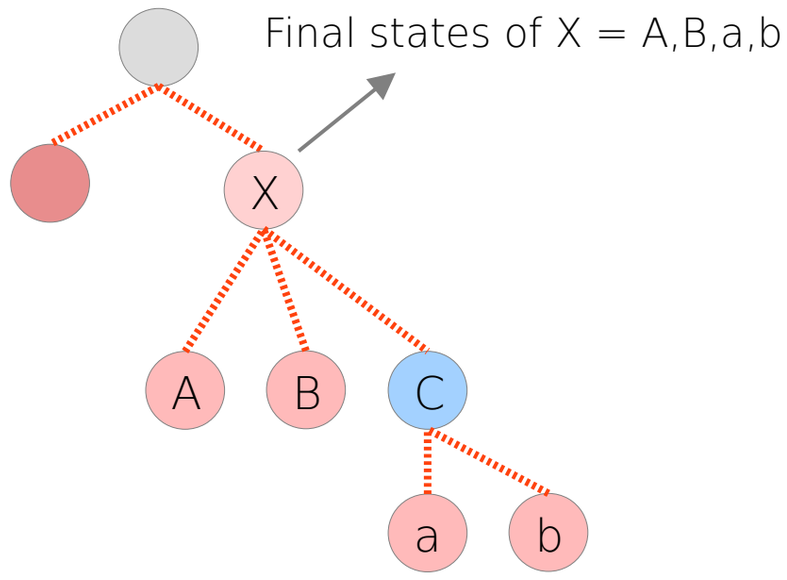
Arbitrary SM final states and BSM states
(InclusiveBranch→ ['*'])

Inclusive Objects – SModelS v3

- In v3 inclusive objects are handled using ***three*** new definitions:

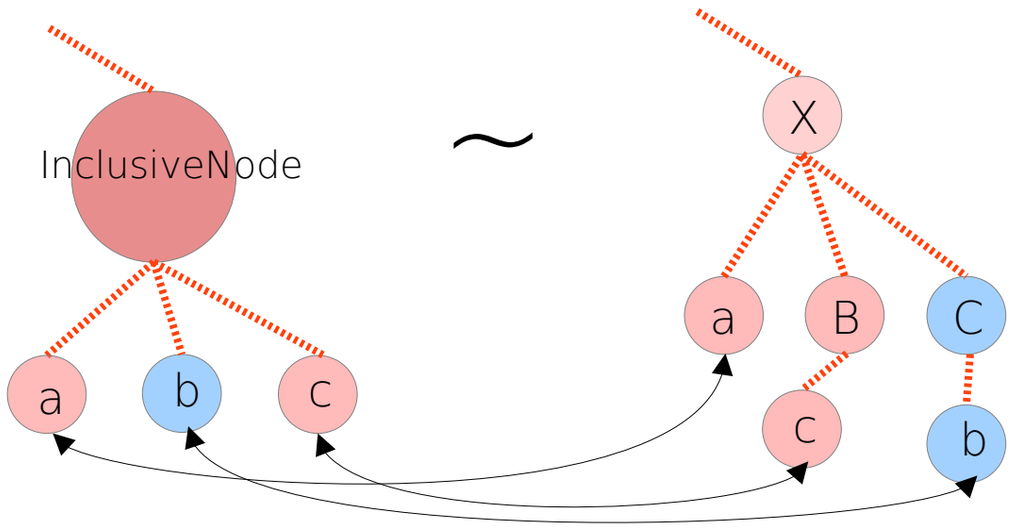
  - <u>Final States of a Node:</u>

    - particles appearing as a final state (undecayed) generated by the cascade decay of the node



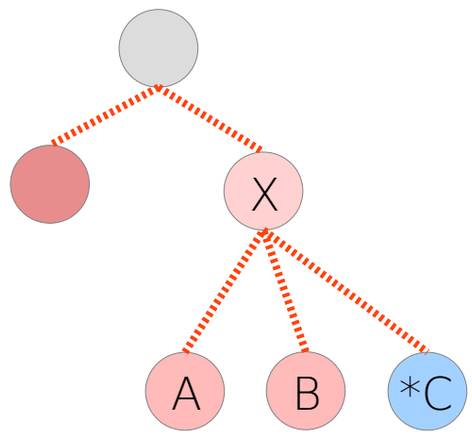
Inclusive Objects – SModelS v3

- Inclusive Node:
  - Its canonName is an inclusive value (*), which matches any int
  - If a tree contains an InclusiveNode, all of its antecessors also have their canonNames set to *
  - It is represented in string form by “Inclusive > …” or by [‘*’] (old bracked notation)
- When comparing to another node, only the ***final states*** of the InclusiveNode and of the other node are compared:

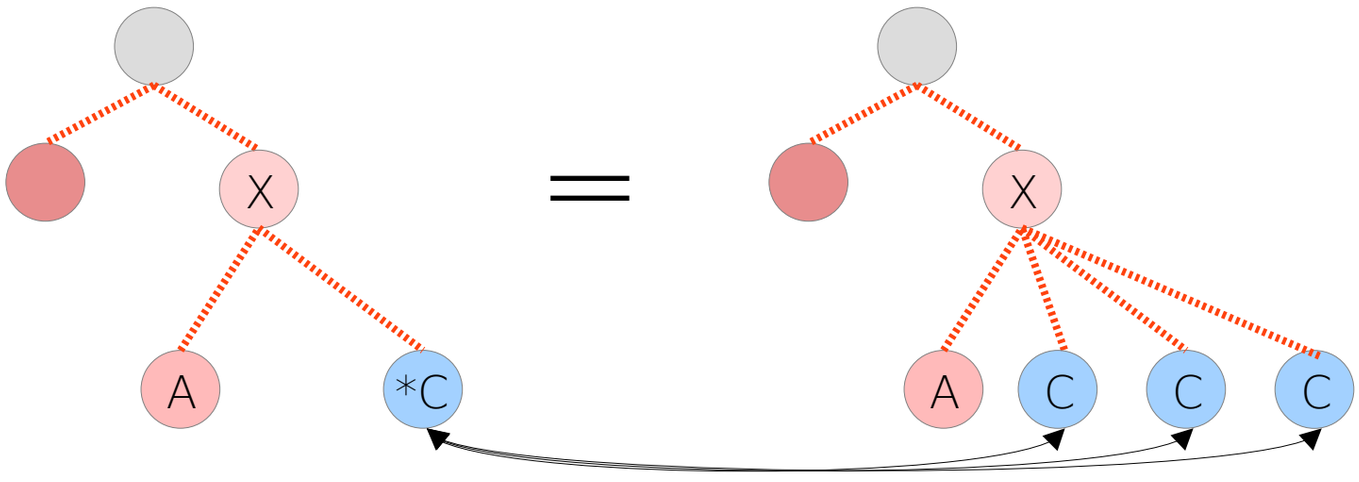


Inclusive Objects – SModelS v3

- InclusiveList Node:
  - Behaves as a regular ParticleNode, however it represents any repeated number of the same particle
  - It is represented in string form by *particleLabel (e.g. *q) → *can only be implemented using the new string format*

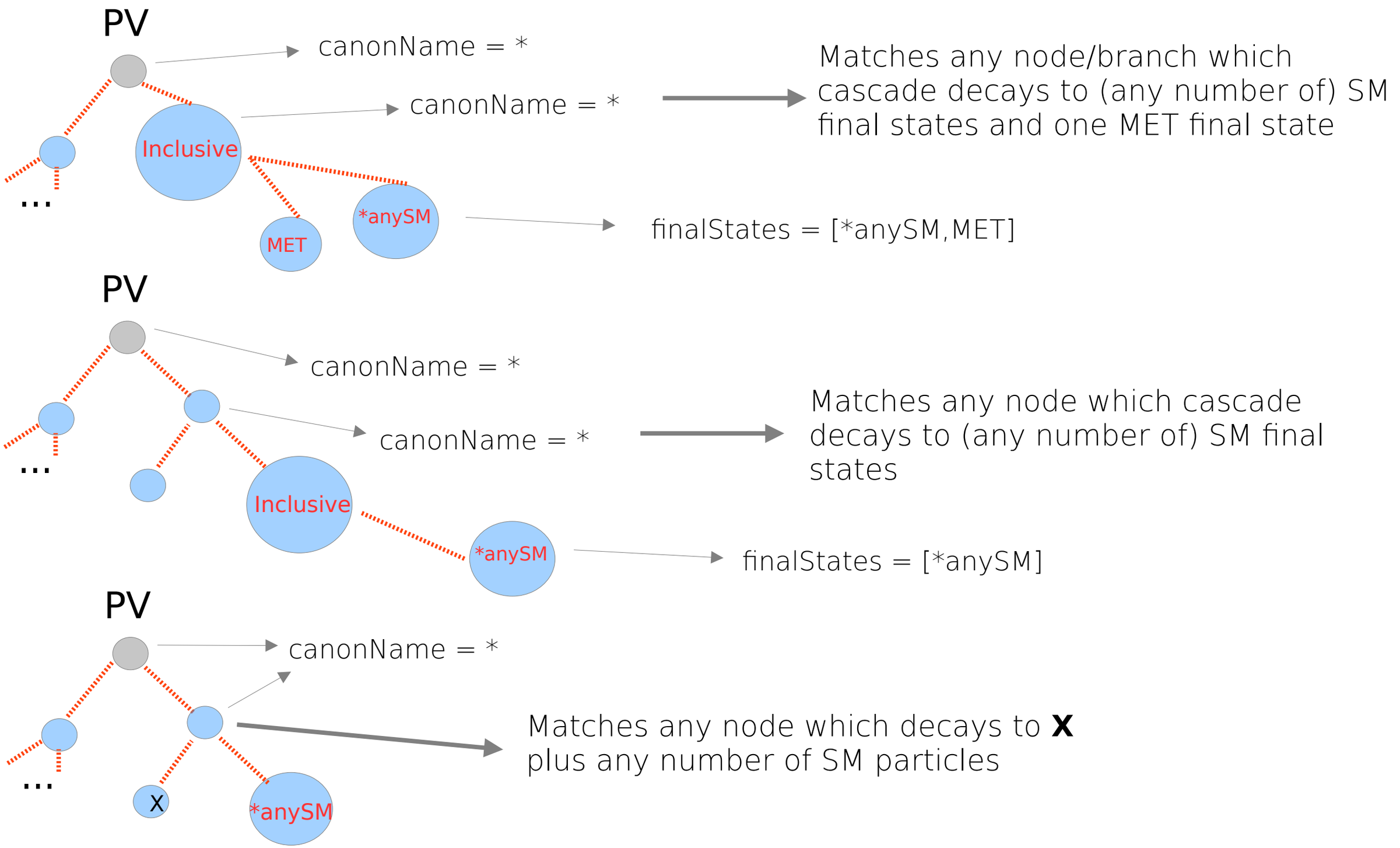


  - When matching to another list of nodes, it will match all the nodes which match its particle:



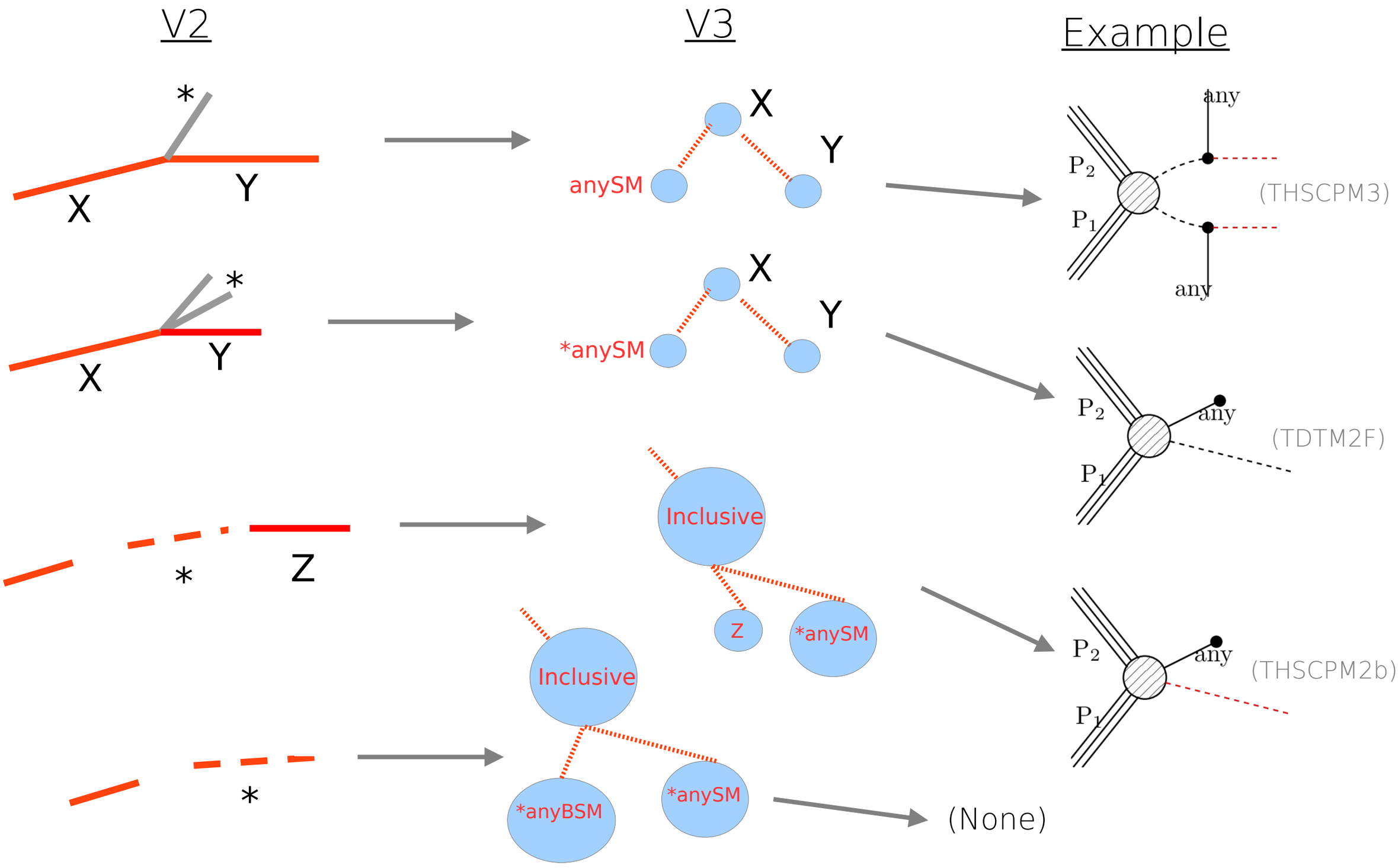
Inclusive Objects – SModelS v3

- Examples:



Inclusive Objects – SModelS v3

- The inclusive objects in v2 correspond to:

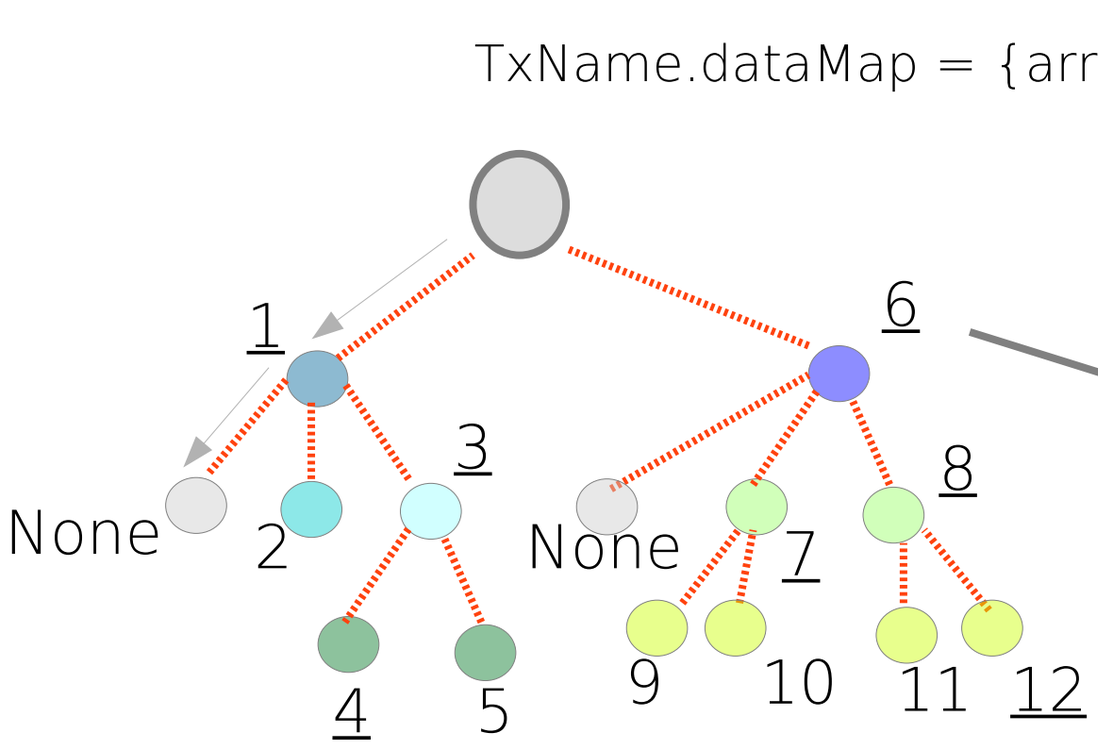


Database Topologies

- How to store information (effs, ULs) for arbitrary topologies?

1)Numbered nodes

2)dataMap → maps the nodes attributes to a flat array



Database - dataMap

- The dataMap holds all the information for extracting the relevant data from a TheorySMS.
- Inclusive nodes are ignored (do not appear in a dataMap)
- When creating a database entry, the dataMap can be directly defined in the TxName.txt file, as well as the flat data points:

constraint : {(PV > anyBSM(1),anyBSM(2)), ...}

dataMap : {arrayIndex : (node number, attribute, unit),...}

efficencyMap: [[[m_1,m_2,...],eff], [[m_1',m_2',...],eff', ...]

or

upperLimits : [[[m_1,m_2,...],ul], [[m_1',m_2',...],ul', ...]

→ flat mass/width data array of floats (without units)

- The node numbers are defined by the particle indices appearing in the constraint:

constraint : {(PV > anyBSM(1),anyBSM(2)), ...}

node = 1 node = 2

Database - dataMap

- For the old format (nested brackets) an intermediate map is constructed, which maps the nested bracket indices to a flat array:

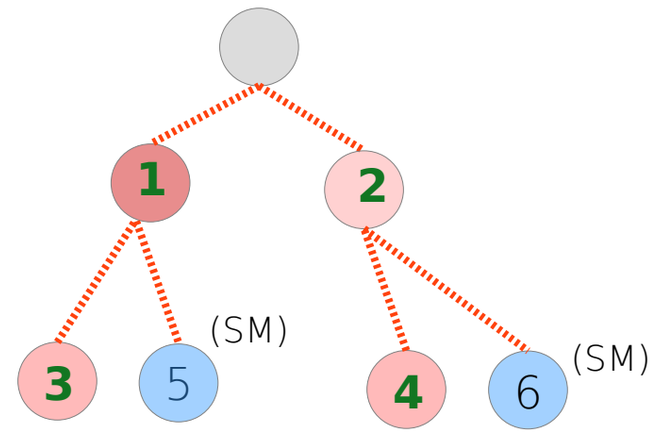
arrayMap : {arrayIndex : ((i,j,k), attribute, unit, node number),…}

flat array index

i,j → indices of nested array
k → 0 for mass, 1 for width (if present in data)

- Example:

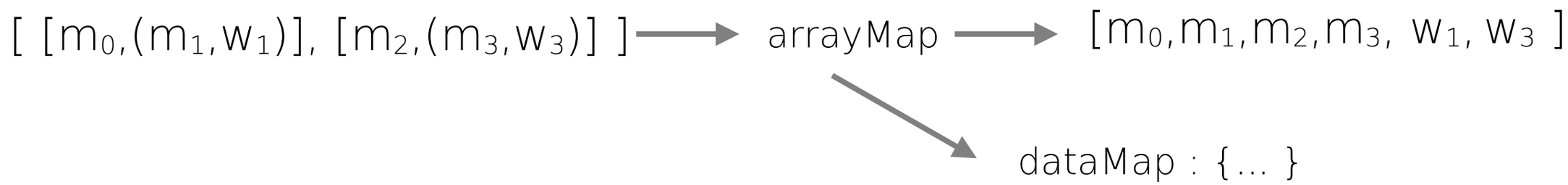
[[m_0,(m_1,w_1)], [m_2,(m_3,w_3)]]



arrayMap : {0 : ((0,0,0), ‘mass’, GeV, **1**), 1 : ((0,1,0), ‘mass’, GeV, **3**),
2 : ((1,0,0),’mass’, GeV, **2**), 3 : ((1,1,0), ‘mass’, GeV, **4**),
4 : ((0,1,1), ‘totalwidth’, GeV, **3**), 5 : ((1,1,1), ‘totalwidth’, GeV, **4**) }

Database - dataMap

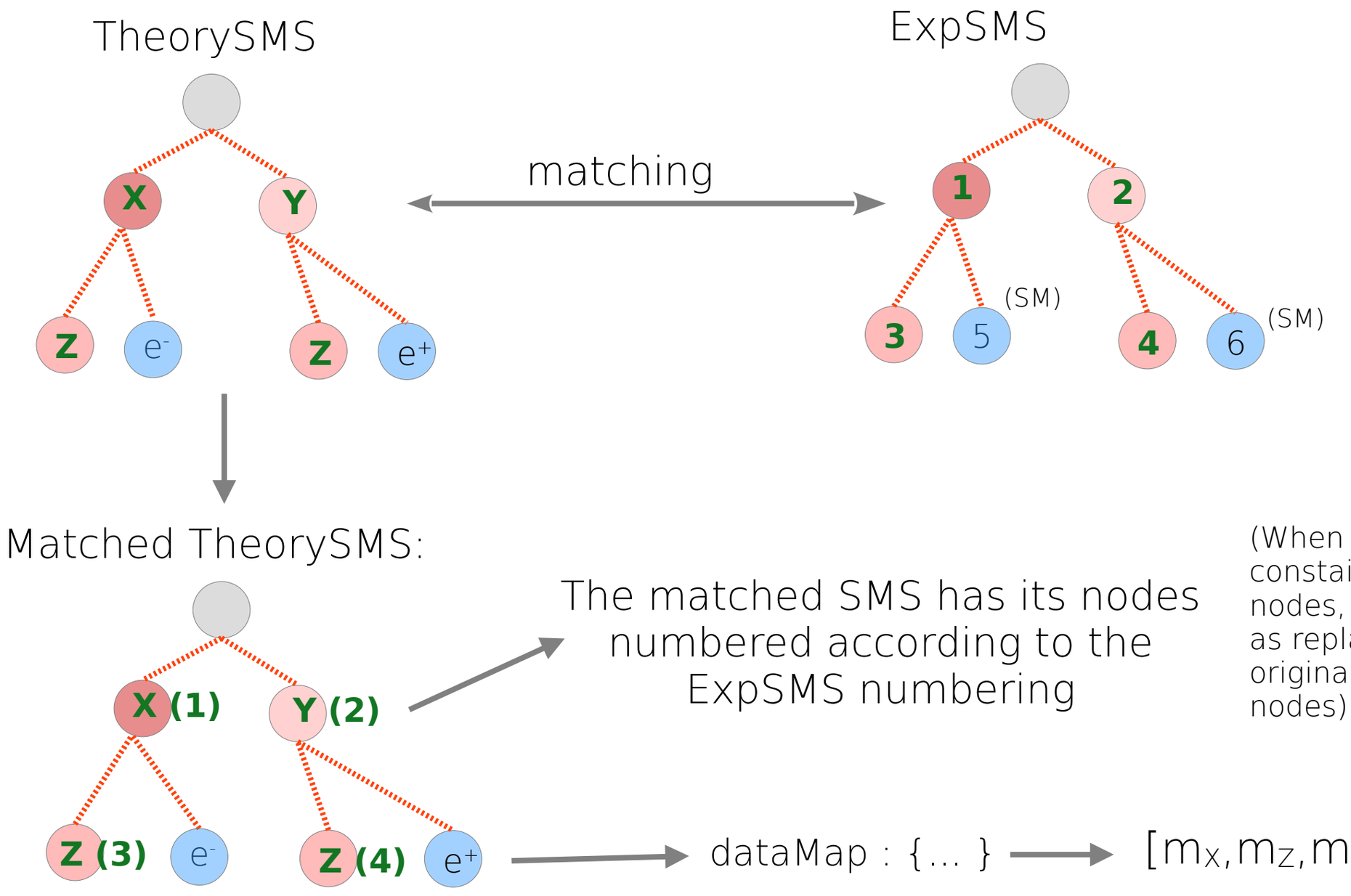
- The arrayMap is used to convert the old data format to a flat array and construct the dataMap:



- Hence the old format can be (internally) converted to a new flat array format with a well defined dataMap

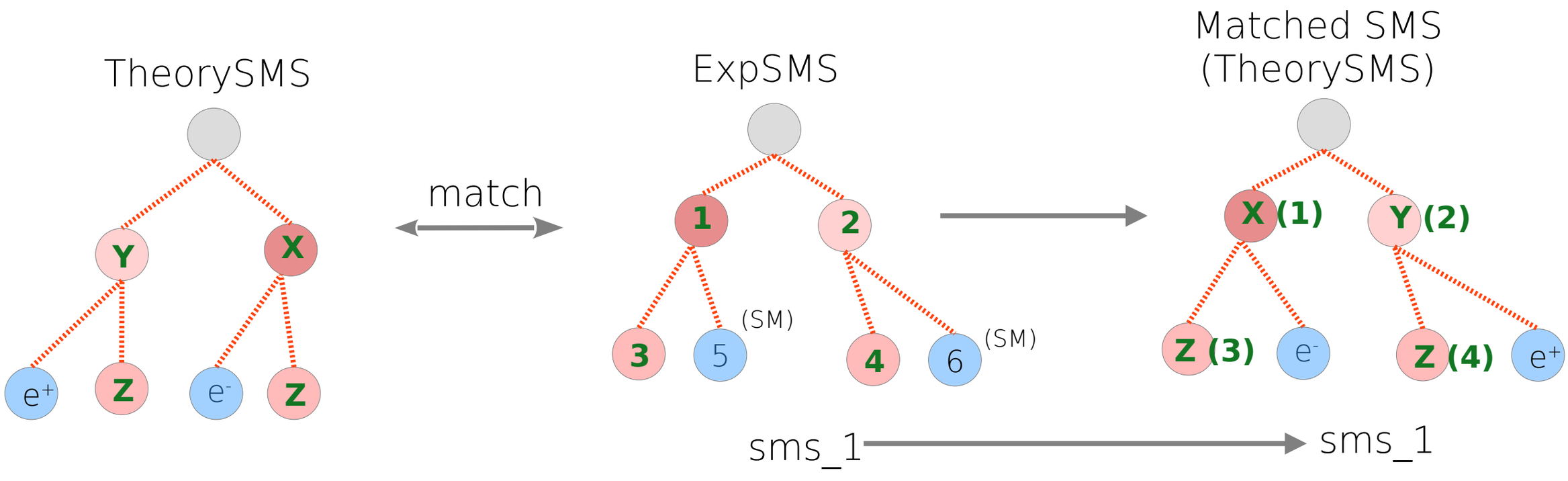
Database - dataMap

- Finally the dataMap can be used for extracting the required information from TheorySMS objects (from the decomposition):



Database – matching

- When matching TxNames with TheorySMS from decomposition:
  - The matching returns the SMS with the same shape and the daugthers ordered in the way they have matched the txname SMS (ExpSMS)
  - The matched SMS nodes are numbered according to the txname SMS numbering
  - The matched SMS receives a label matching the label used for evaluating constraints.



Database – Constraints and Conditions

- The constraints and conditions are string expressions for the SMS weights:

constraint: 2*({(PV > …} + { (PV > …)}) → v3 format

or

constraint: 2*([[[…]]]+ [[[…]]]) → v2 format

condition: Csim({(PV > …},{ (PV > …)}) → v3 format

or

condition: Csim([[[…]]], [[[…]]]) → v2 format

- *Note that in v3 strings representing SMS are delimited by curly brackets:* {(PV > …} = {ExpSMS}
- *All nodes appearing in the SMS description should be numbered and match the dataMap*

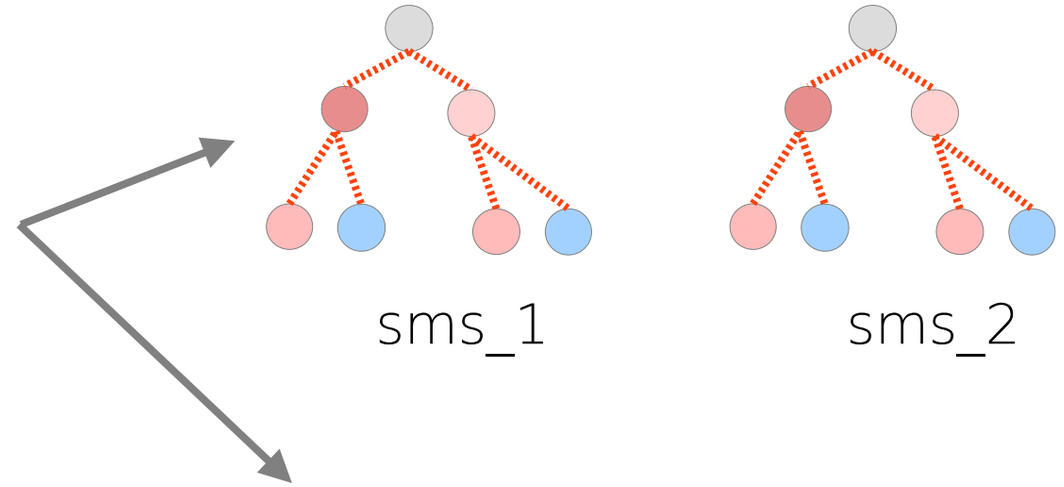
Database – Constraints

- In v3 the SMS appearing within the constraints are converted to ExpSMS objects and receive labels, which are used as replacements in the string expressions. The constraint function (string) is also created using the labels:

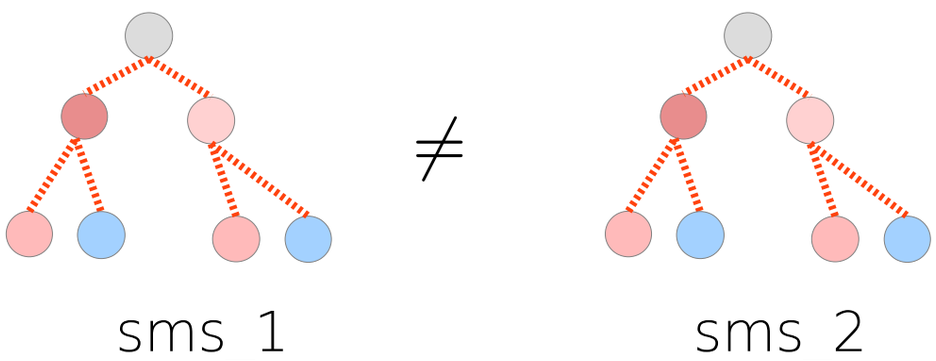
constraint: 2*({(PV > …} + { (PV > …)})

or

constraint: 2*([[[…]]]+ [[[…]]])



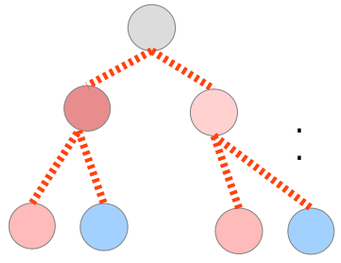
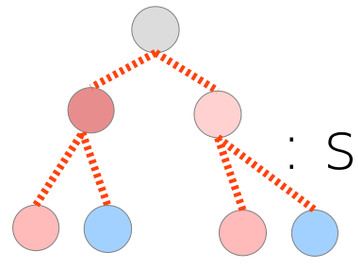
- The SMS appearing in the constraints are required to be unique, which means that distinct ExpSMS belonging to the same TxName can not mach:



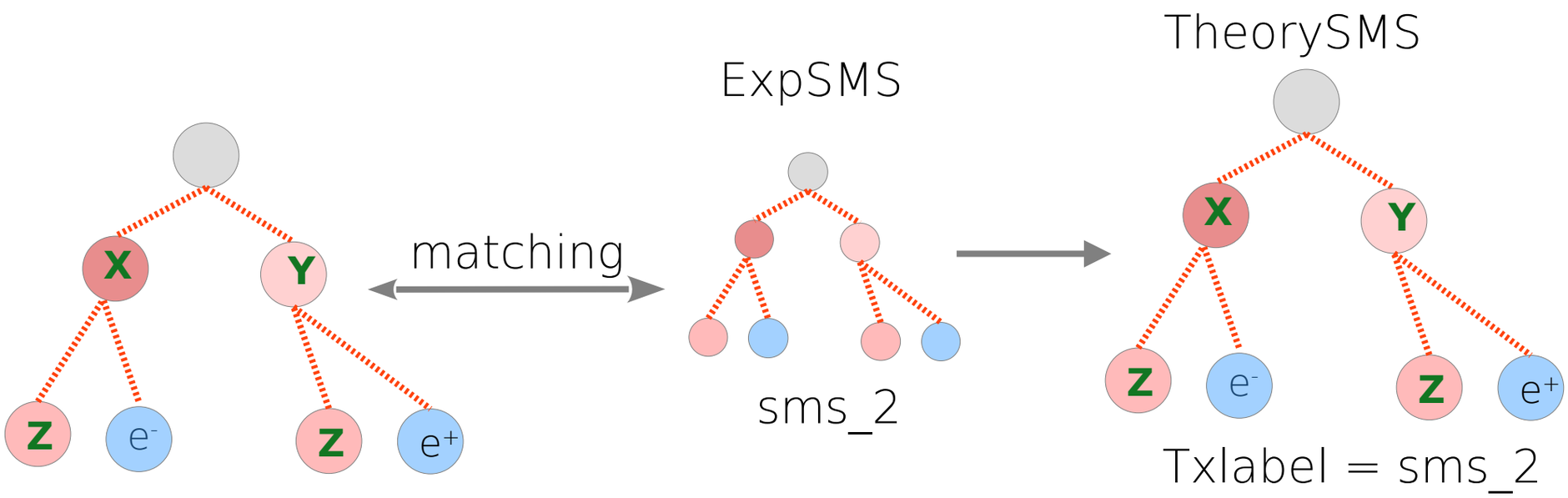
- This restriction is needed in order to make sure that the same TheorySMS will not match more than one ExpSMS in the same constraint (to avoid double counting)

Database – Constraints

- The SMS and respective labels appearing in constraints are stored in the TxName.smsMap:

constraint ⟶ smsMap = {  : sms_1,  : sms_2, …

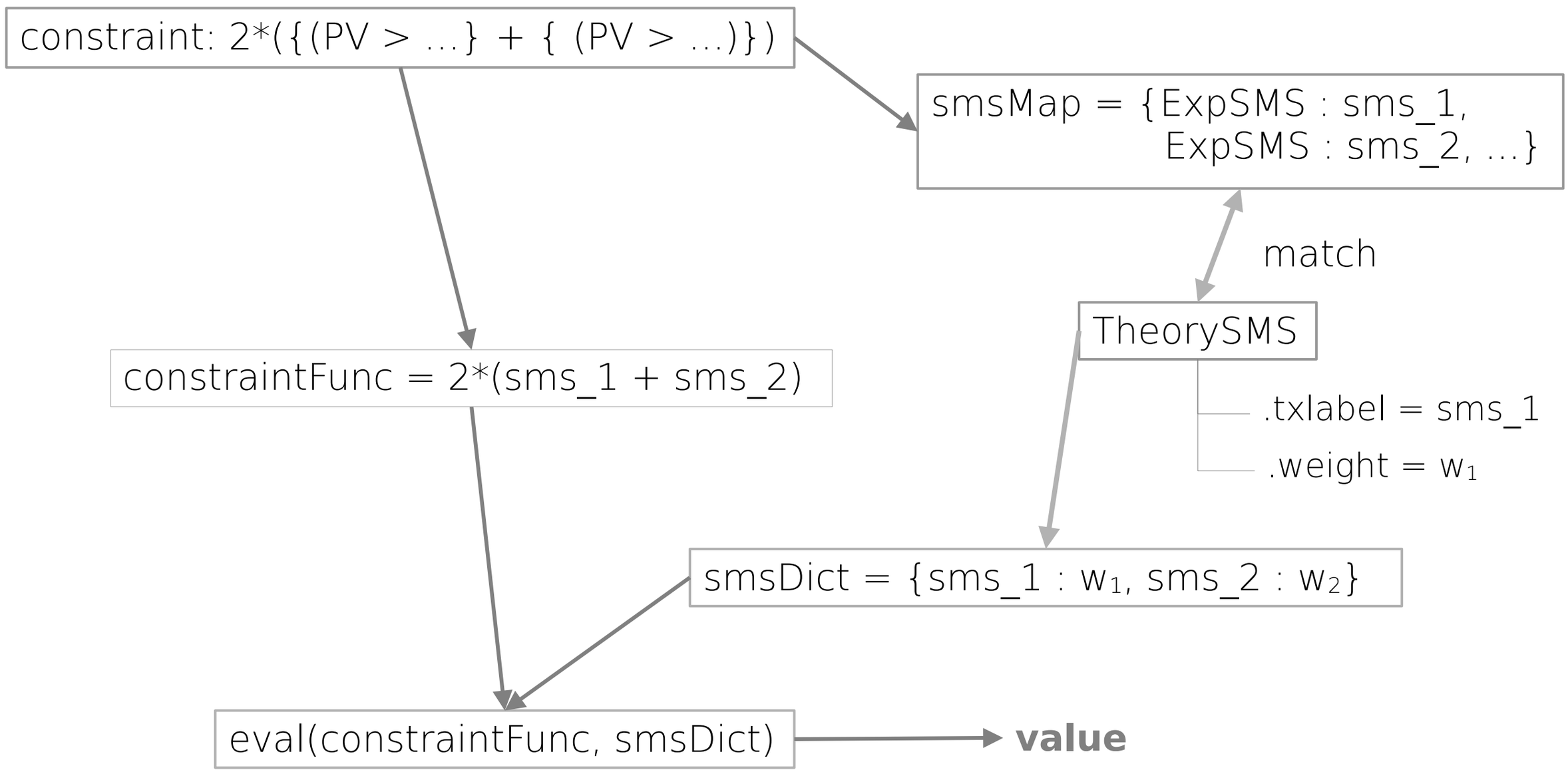
- When checking if a TheorySMS appears in the TxName, the ExpSMS objects from smsMap are used for comparison. Since the TheorySMS has to match a ***single*** ExpSMS, it is "tagged" with the ExpSMS label:



the txlabel is then used to evaluate the constraints using the constraintFunc

Database – Constraints

- The constraints can then be evaluated using a dictionary mapping the TheorySMS weights to their txlabels.
- In summary:



Database – Conditions

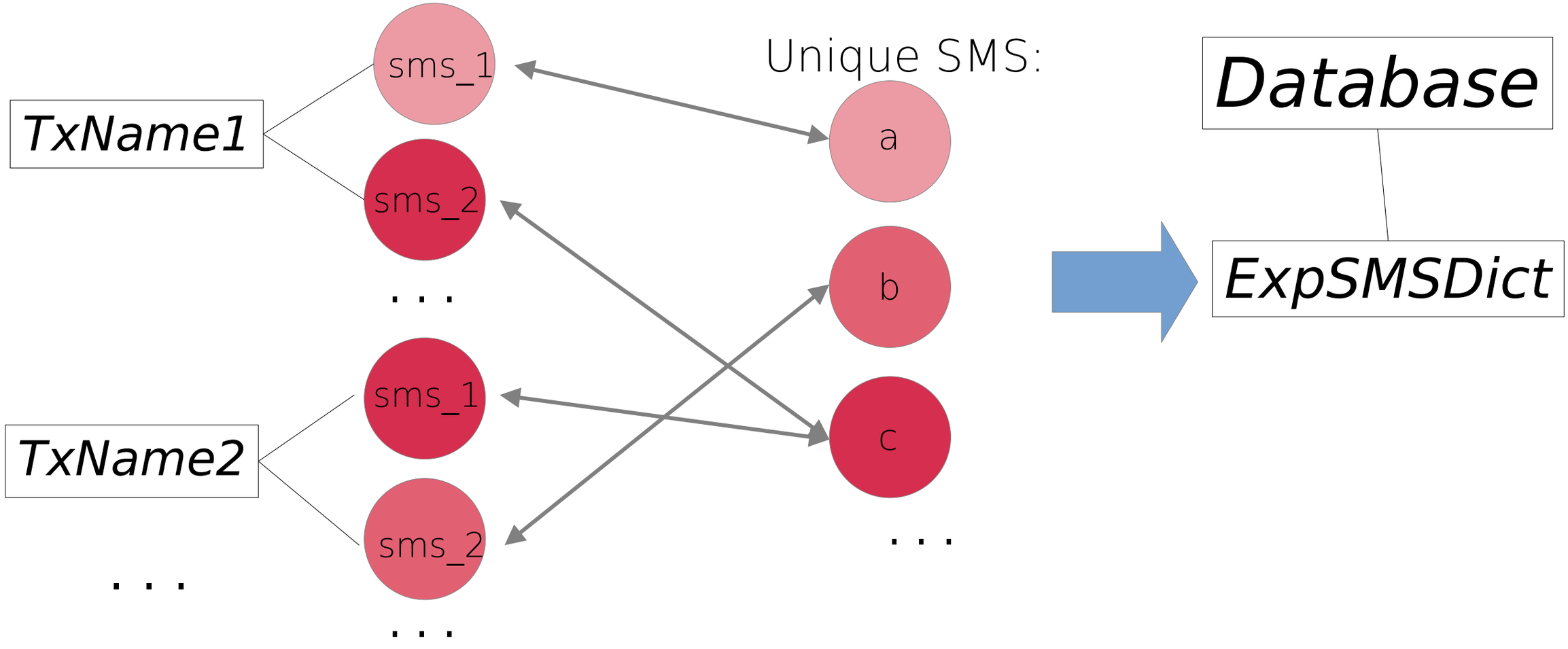
- Evaluating conditions are less straightforward, since they may contain overlapping SMS. For instance, in the example below the first SMS ([[[L,L]],[[L,nu]]]) contains the second:

conditions: Cgtr([[[L,L]],[[L,nu]]], 3*[[[L,L]],[[ta,nu]]]); Csim([[[L,L]],[[e,nu]]], [[[L,L]],[[mu,nu]]])

- Therefore, for conditions, a smsMap is defined for each condition and the map can contain overlapping SMS.
- In this case the TheorySMS can match multiple SMS and the ordering of each matched SMS does not have to be the same.
- Therefore, when evaluating conditions, the SMS matched by the constraints are compared to the ones contained in conditions. If it matches, its weight is added to the matched sms label.

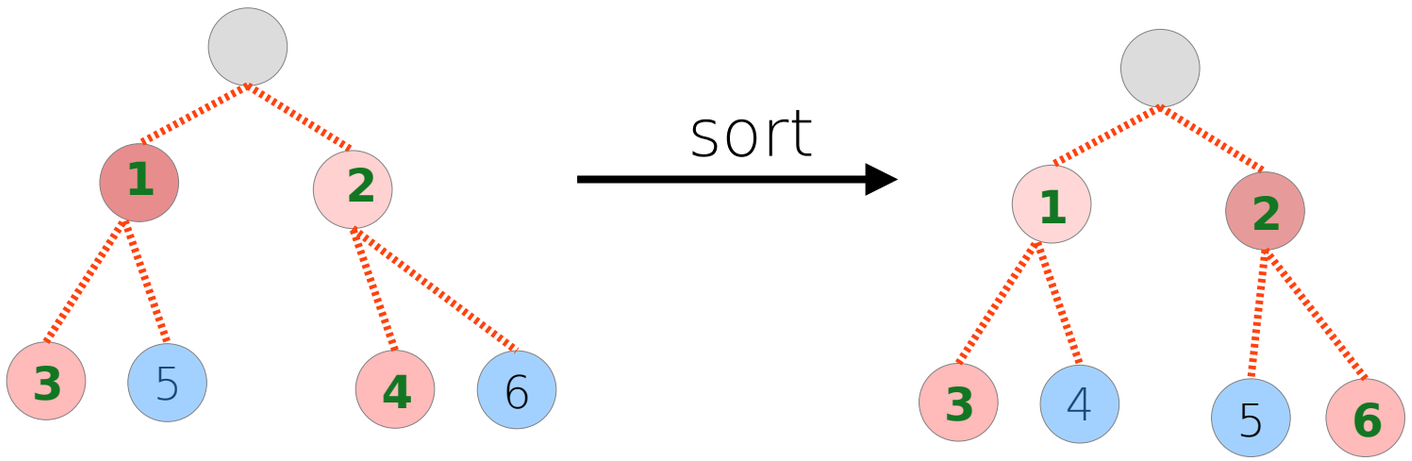
Database – ExpSMSDict

- In order to enhance the performance for the calculation of theory predictions and avoid duplicated comparisons, the matching between TheorySMS and ExpSMS is done in a centralized fashion.
- First, all the ExpSMS in the database are collected and a unique list of (sorted) ExpSMS objects is built using the ExpSMSDict class. This class holds a mapping between the unique SMS and the TxNames they belong to:



Database – ExpSMSDict

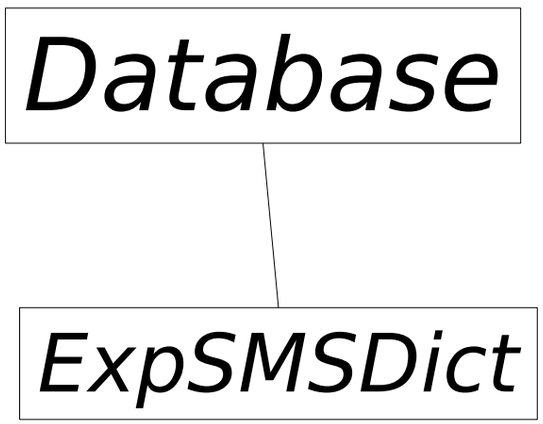
- Since the unique SMS are sorted, the ExpSMSDict object also holds the mapping between the numbering of the sorted nodes in the unique SMS and the node numbering in the (original) txname SMS:



NodesDict = {1 : 2, 2 : 1, 5 : 5, 6 : 3, 3 : 4, 4 : 5}

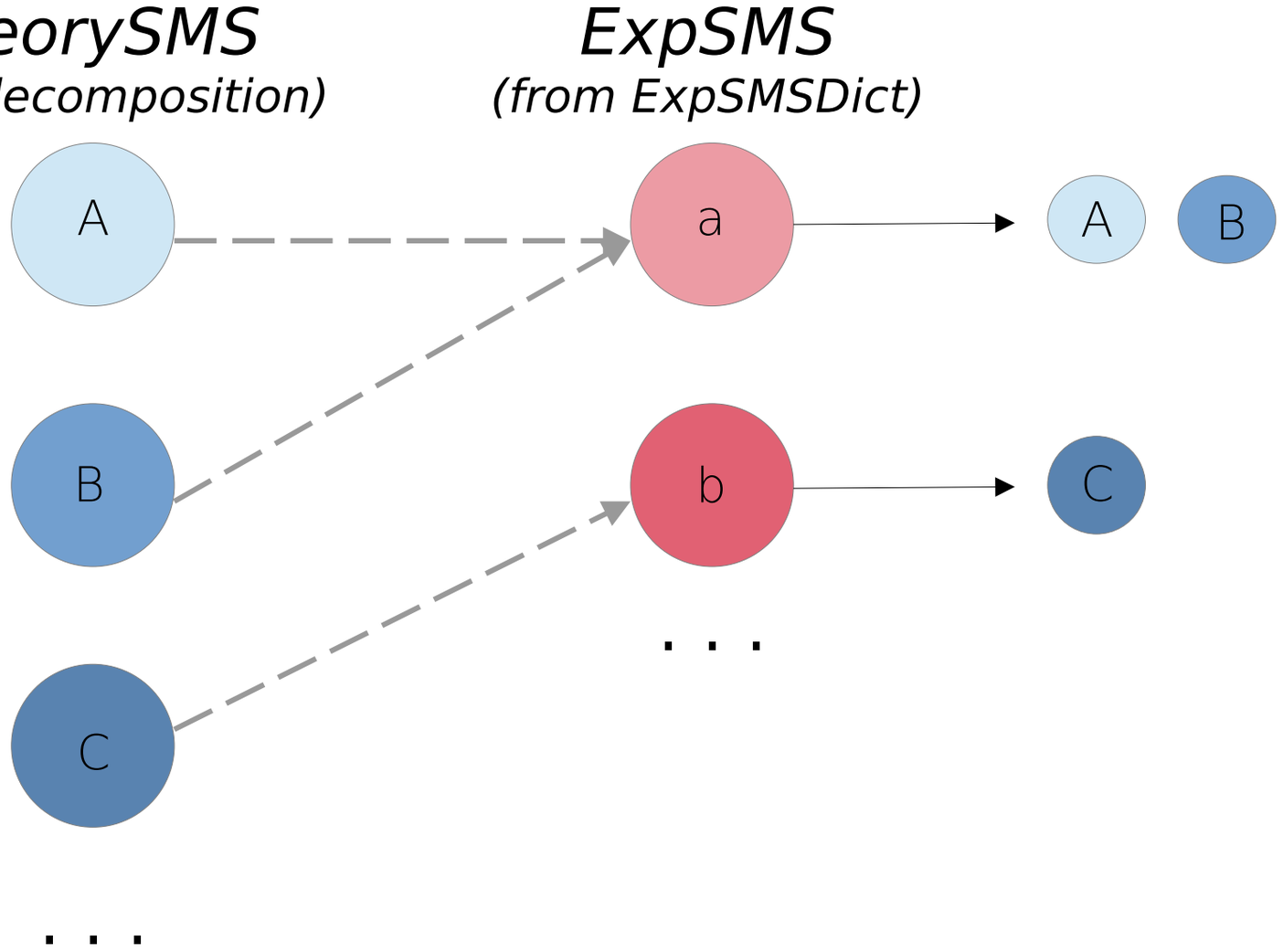
new index (unique SMS)

old index (txname SMS)



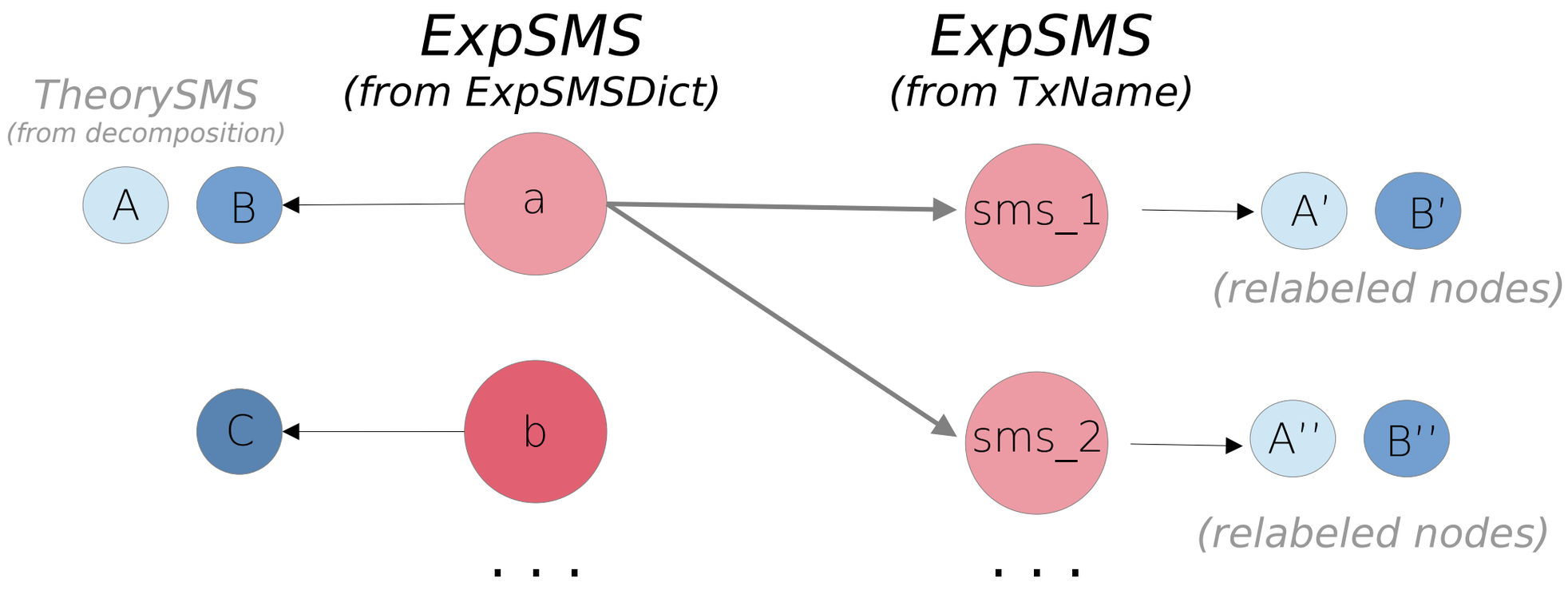
Database – ExpSMSDict

- The list of unique SMS are then used to match the ExpSMS to the TheorySMS from decomposition. Since the number of unique SMS is much smaller than the (duplicated) txname SMS the number of comparisons between TheorySMS and ExpSMS are greatly reduced.



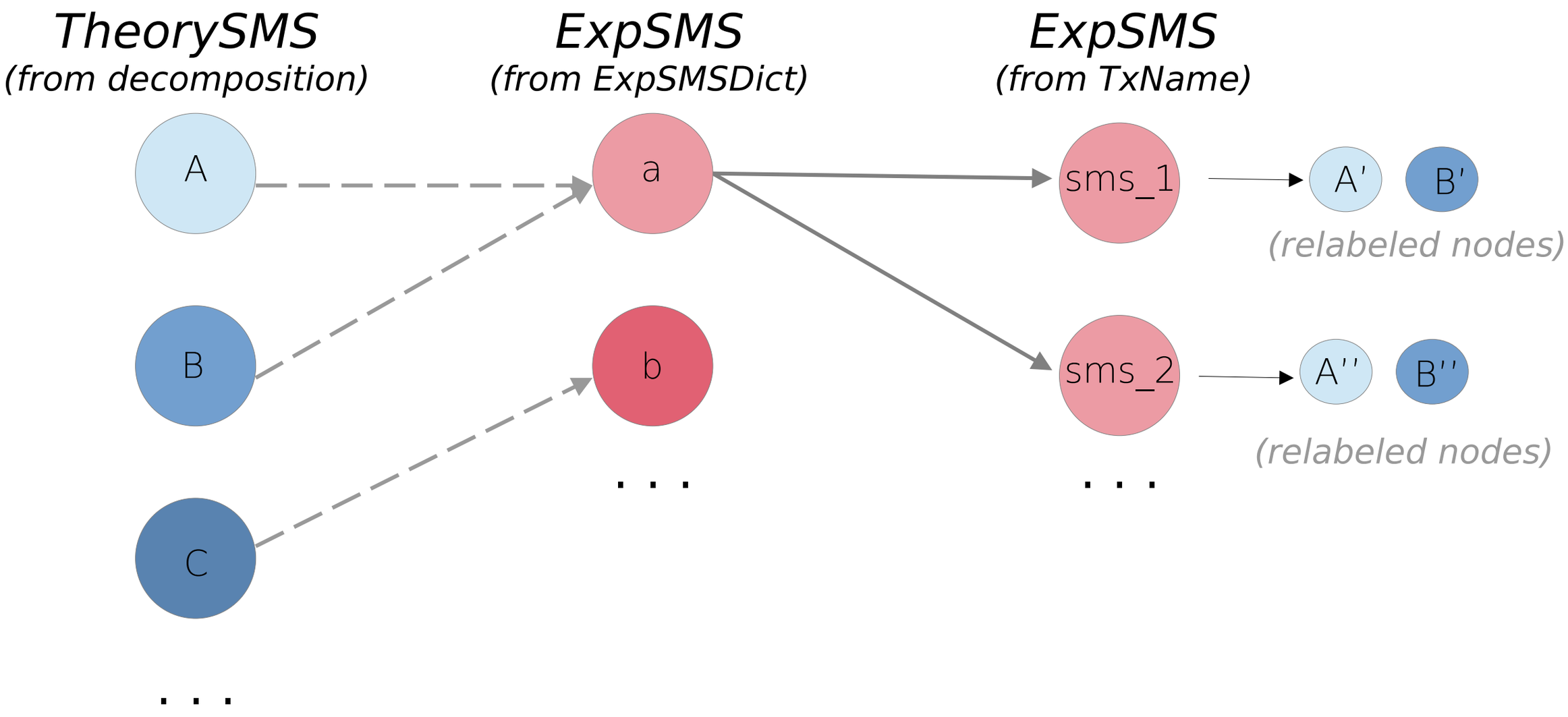
Database – ExpSMSDict

- Finally, when computing predictions for a specific dataset/TxName, the following steps are taken:
  1. The ExpSMSDict is used to fetch the unique SMS corresponding to a given TxName SMS
  2. The TheorySMS which were previously matched to the unique SMS are then copied and passed on to the TxName SMS
  3. The TheorySMS nodes are relabeled to correspond to the TxName SMS labeling (using the nodesDict from ExpSMSDict)



Database – ExpSMSDict

- In summary, the matching between decomposition (TheorySMS) SMS and the database (ExpSMS) is done through the unique SMS stored in ExpSMSDict:



Conclusions

- The implementation is somewhat encapsulated (it does not affect the higher levels of the code)
- Graphs can be an useful tool
- The main reamining issue is how to resolve ambiguities for equivalent nodes
- Some functionality is already in place in smodels:graphs