documentation

- Research reports
- Musical works
- Software

# OpenMusic

## RC library

version 1.1

# Tutorial

*First edition, March 2000*

ircam
Centre
Pompidou

This tutorial was written by Örjan Sandred, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

OpenMusic was conceived and programmed by
Gérard Assayag and Carlos Agon.

The RC library was conceived and programmed by Örjan Sandred.

First edition of the tutorial, March 2000.

This tutorial corresponds to version 1.1 of the RC library, and to version 2.0 or higher of OpenMusic.



**Ircam**
**1, place Igor-Stravinsky**
**F-75004 Paris**
**Tel. 01 44 78 49 62**
**Fax 01 44 78 15 40**
**E-mail ircam-doc@ircam.fr**

# IRCAM Users' group

The use of this software and its documentation is restricted to members of the Ircam software users' group. For any supplementary information, contact:

Département de la Valorisation
Ircam
Place Stravinsky, F-75004 Paris

Tel. 01 44 78 49 62
Fax 01 44 78 15 40
E-mail: bousac@ircam.fr

Send comments or suggestions to the editor:
E-mail: bam@ircam.fr
Mail: Marc Battier,
Ircam, Département de la Valorisation
Place Stravinsky, F-75004 Paris

http://www.ircam.fr/forumnet

# Tutorials for the RC library

The goal with the tutorials is to illustrate the most important features of the RC library. The examples try to be as clear and simple as possible. To illustrate finished, musical interesting examples have had a low priority. It is up to the user to do this in his own work. It should be said, with emphasis, that the goal with the RC library is not to create a machine that fast generates a good result, but to build a platform for composers to work with rules on rhythms. Some suggestions on how to confront the problem is given, but the most important part is for the user to invent and build his own rules. To find a good result takes some experimentation on finding both the right type of domain as well as rules. Sometimes very small changes can have a great impact on the result.

The example patches are provided along with the library. To use them, drag the folder "RC examples" from the folder "User Library:OMRC 1.1" to the OpenMusic Workspace window. Double-click on the patch icons: the OMRC library and other required libraries will be loaded automatically.

## 0. The engine

The heart for all calculations in the RC library is the search engine. There are two different search engines in the Open Music environment, the *pmc* in the "OMCS" library (transferred from the "PWConstraints" library) by Mikael Laursen, and the *Csolver* in the "Situation" library by Camillo Rueda. They do not work in an identical way, and they use different formats for the input data.

The RC library supports both engines. However, the *pmc* has proven itself to be the most efficient in connection with this library. It is also the only one that can handle heuristic rules (see paragraph 6 below). If the user wants to work with the *Csolver*, he should read paragraph 9 below.

The search engine is not part of the RC library. To use the *pmc* engine, charge the "OMCS" library. To use the *Csolver* engine, charge the "Situation" library.

## 1. The domain

A domain is a set of possible answers to a problem. The search engine will pick elements from the domain, and check if they are true according to rules given by the user. The output from the search engine is a sequence of elements ("variables") from the domain that all passed the rules as true.

Each element in the domain is in the RC library either a single note value, a rhythm motif, or a time signature. The element is the smallest building block in the answer, and it can not be transformed, neither split up into smaller elements.

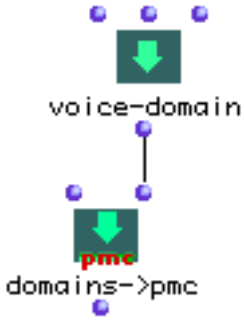



*Fig. 1.*

The *domains->pmc* formats the domain for the pmc engine. The *voice-domain* is connected to the second input on the *domains->pmc*, and the first input is the number of variables the search engine will output in the solution (how long the sequence of elements will be).

The first input on the *voice-domain* is always reserved for time signatures (not used in the first examples). The second input is a domain of note values, or rhythm motifs (or mixed).

The format the search engine outputs is not directly readable (you can easily see this output in the listener window if you evaluate tutorial-01a directly on the *pmc*). This is why the function *decode-engine* is necessary. Evaluate the patch in tutorial-01a a few times. Change the left input on the *domains->pmc* and notice the difference in the result.

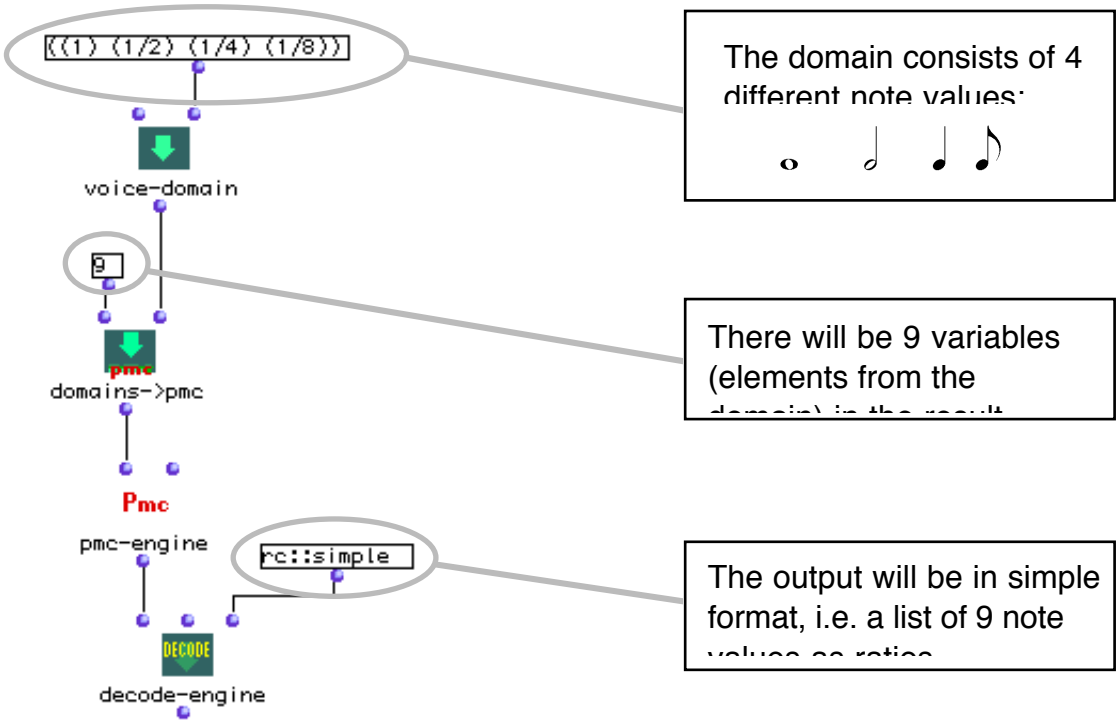



*Fig. 2. Tutorial-01a*

The output format from *decode-engine* can be changed in the menu in the third input (the box has to be expanded first). If “voice-list” is chosen, the output can be sent directly to a *poly-object* (see tutorial-01b, fig.3). The tempo used in a score object can be set in the second input on the *decode-engine* (it will affect MIDI playback).

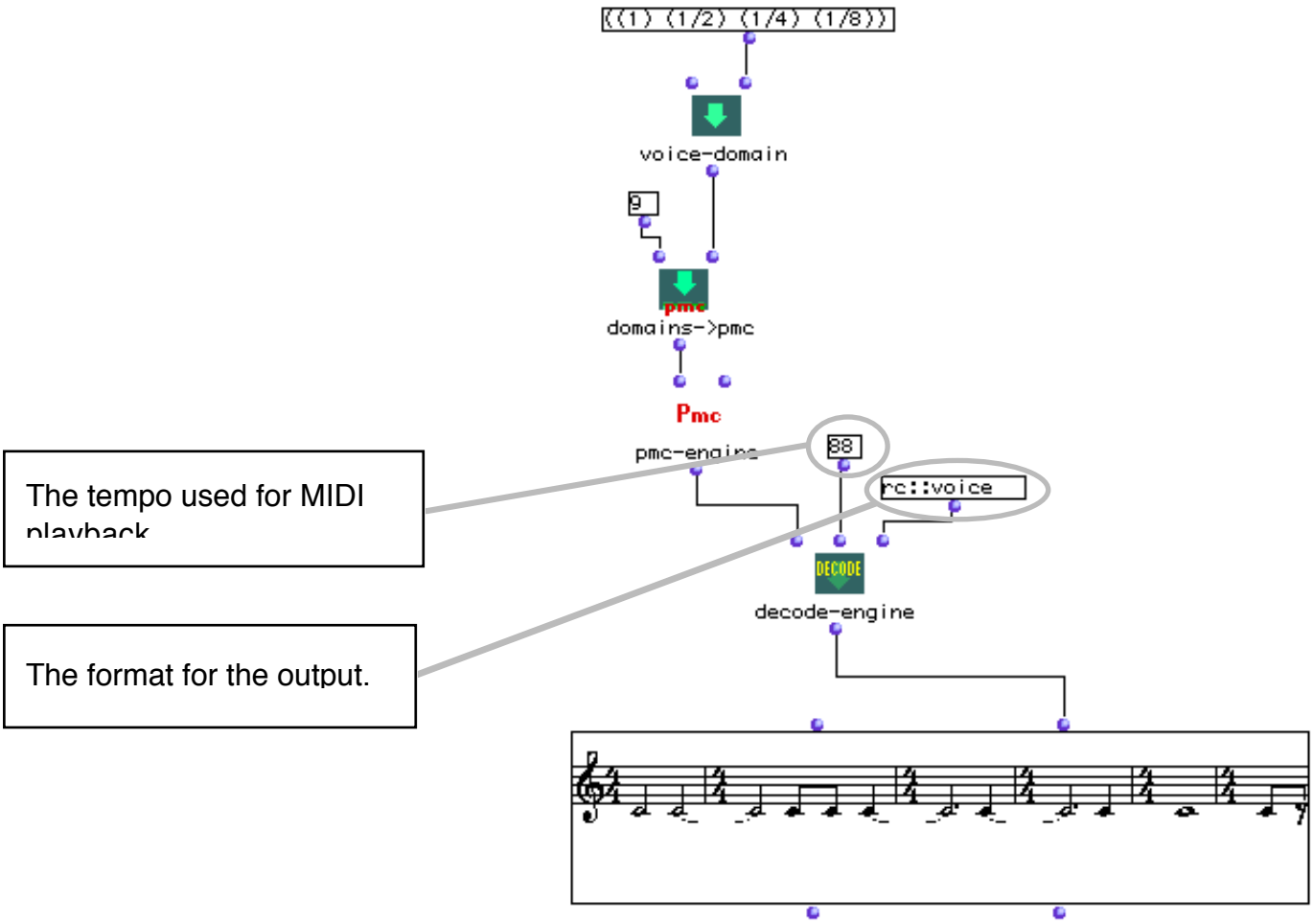



*Fig. 3. Tutorial-01b*

The *voice-domain* box can also be expanded. Each input, with the exception of the first one, represents a layer in the answer. Two or more voices can be calculated at the same time (and affect each other). Each layer can have its own, separate set of note values/rhythm motifs. The search engine keeps track on which layer an element belongs to. The *decode-engine* also understands this, and outputs the elements in the correct layers.

Study tutorial-01c, fig.4. Since there are no rules, the search engine can choose randomly from the domain. Because of this, the length of each layer can be very different. It is even possible to get a solution that only has elements in one layer.

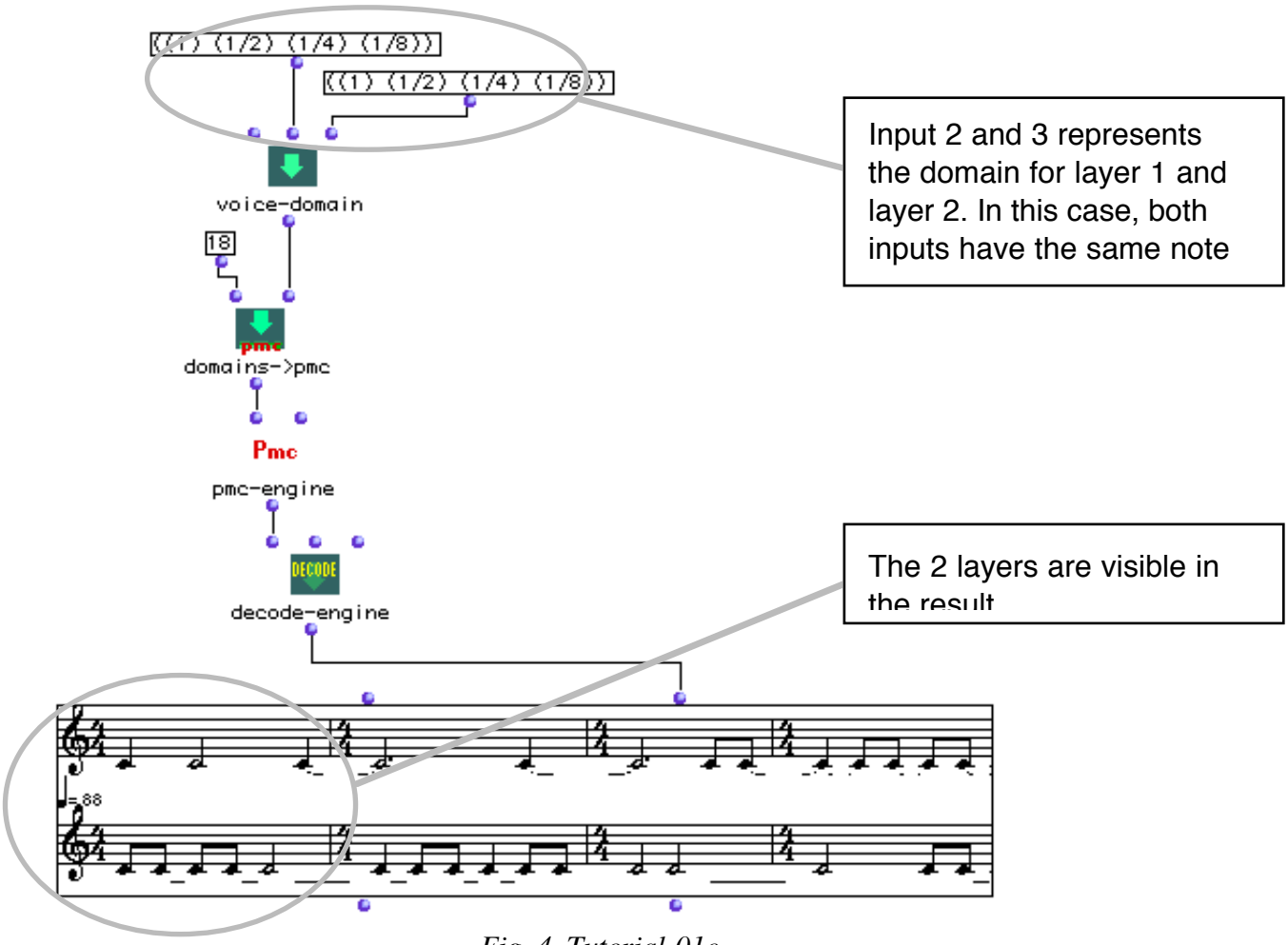



*Fig. 4. Tutorial-01c*

# 2. The rules

In the same way there is an interface for formatting the domain for the search engine, there is an interface for formatting the rules: *rules->pmc*. There are a few standard rules in the library. Some of them are there for compositional reasons, and some are there to help the search engine find a reasonable solution. The rule in the example in fig.5 is an example of the latter.

The rule *r-eqlength* checks (for each element) which one of the two layers that is the shortest at the moment (during the search). It will only accept an element belonging to the for the moment shortest layer. Because of this the search engine will jump between the layers in a type of "zig-zag" way, when build the resulting sequence.

If this rule is not used, it is often a very high risk that the layers in the answer will be of very different length.

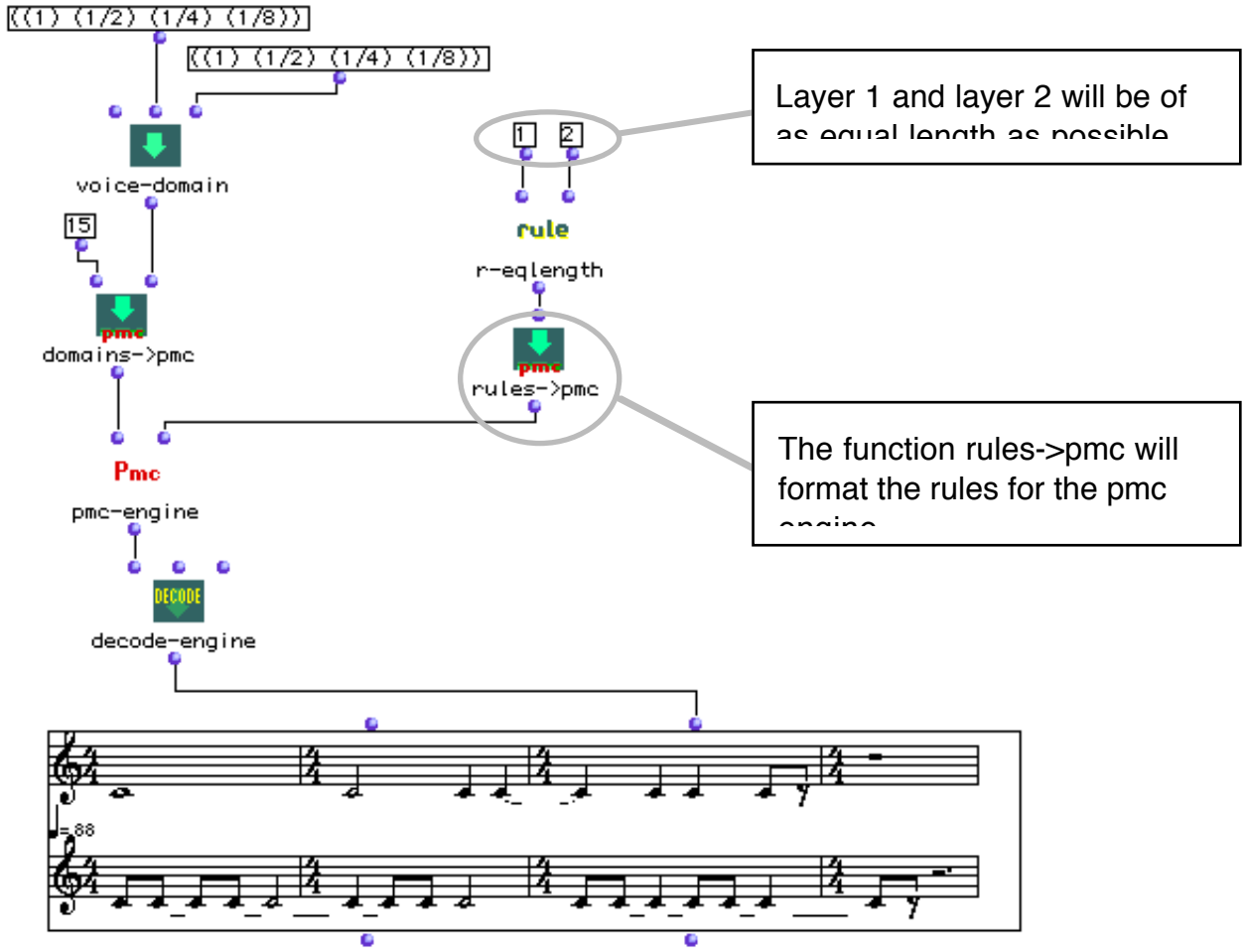



*Fig. 5. Tutorial-02a*

When using more than one rule, you have to collect them all in one list before you input them to the *rules->pmc*. You collect them with a standard Lisp function, *append* (in the menu Kernel/Lisp).

A compositional rule is added in tutorial-02b; *r-canon*. This rule will build a rhythmical canon between two layers. In the example in fig.6, the second layer has the same rhythm as the first layer, starting one half note later (you can easily confirm this in the solution). The first half note in the second voice is not controlled by this rule.

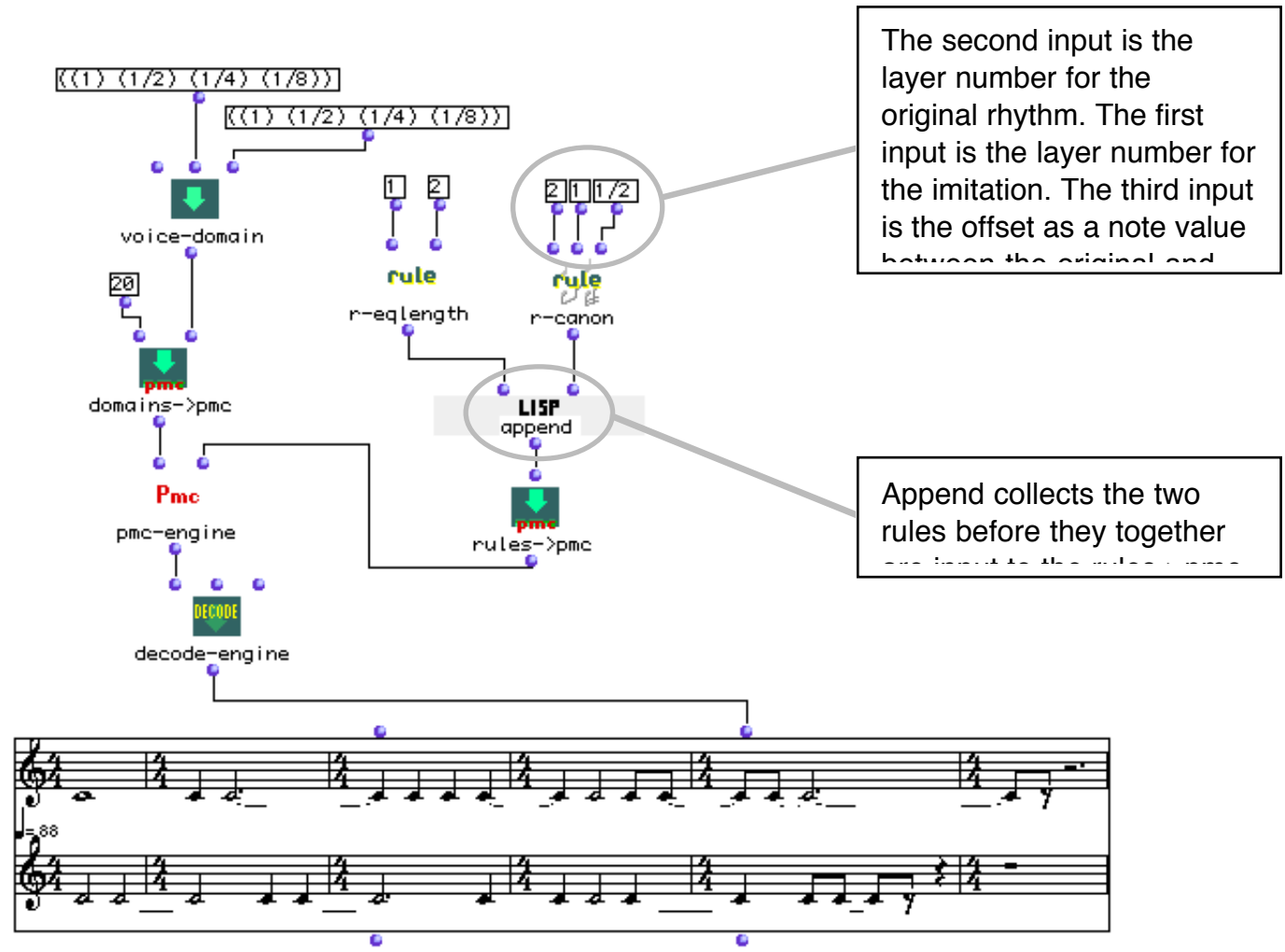



*Fig. 6. Tutorial-02b*

# 3. Time signatures

If a composer considers metrical hierarchy in his music, the positions for events in measures are important. Events on beats are experienced differently from events on syncopations. There are rules to control the relation between rhythm events, time signatures and beats in the RC library. You can divide a rhythm sequence into bars according to rules, or you can fill existing measures with events according to rules, or use a dynamic combination of the two.

The first input on the *voice-domain* will now be used: this is where the set of time signatures is input. The engine can only choose between these. Time signatures are treated as a separate layer by the search engine, and this is always layer number 0.

The rule *r-eqlength* will also be useful in connection with time signatures: we want the sequence of time signatures to be of the same length as the rhythm sequence. If the sequence of time signatures is shorter than the rhythm sequence, the *decode-engine* will stop notate the sequence when all time signatures are used (if a time signature domain is not used, the *decode-engine* will use 4/4 for as many measures as necessary).

The example in fig.7 only use the rule *r-eqlength*. There is no control of the relation between the events and the time signatures. As a result, the notation is very complex: sixteenth note quintuplets occur on syncopations, etc. The output from the *decode-engine* might even be so complex that the score objects in OpenMusic are not able to notate it (however the RC library can work with any degree of complexity). If you get this error message (something like “Error while evaluating the box poly”), just evaluate the patch again.

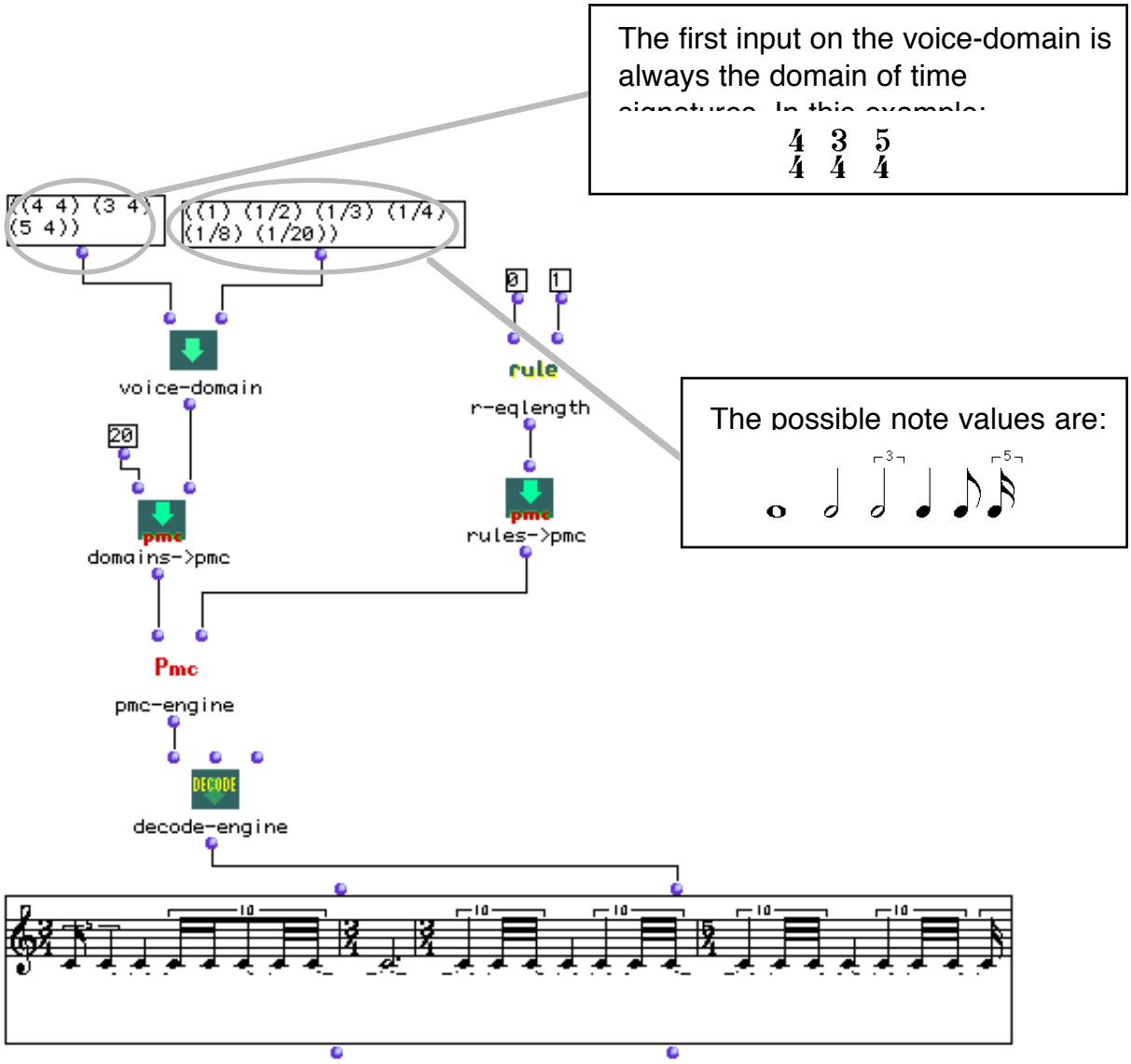
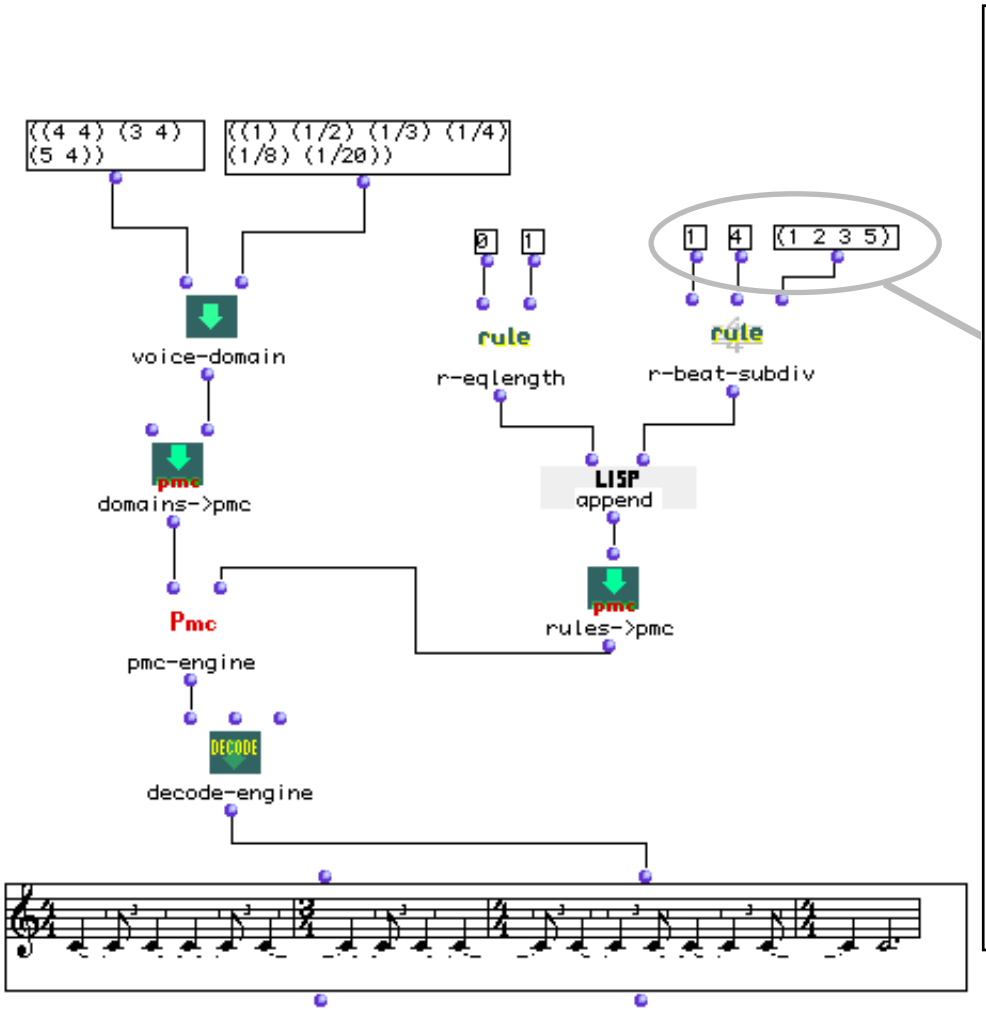



*Fig. 7. Tutorial-03a*

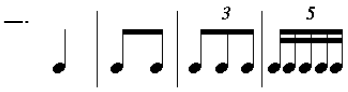
In tutorial-03b (fig.8) a rule for controlling how beats in measures can be subdivided is added; *r-beat-subdiv*. The rule checks how events are positioned in relation to beats, and only allow a limited number of possibilities. The most strict possibility is to only allow events to start on beats. In that case all note values shorter than the beat will be forbidden, as well as longer note values only are allowed to start on a beat (and not on syncopation).

The patch in tutorial-03b should now never give error messages anymore. The rule *r-beat-subdiv* will not allow any complexity in the notation.




The first input is the number for the rhythm layer the rule is checking towards the time signatures.

The second input is the length of the beat in the time signature the rule is checking (i.e. the lower number in the time signature). In this example the rule is only checking the relation between rhythm layer number 1 and time signatures with the beat length one quarter note (4).




Se also fig.9 below.

*Fig. 8. Tutorial-03b*

In fig.9 three examples are given to illustrate how the rule *r-beat-subdiv* works (these examples do not use the same domain as the example in fig.8). In example I all events start on beats, and the rhythm would pass the setting on the rule r-beat-subdiv as shown to the right. The setting in example II allows events to start on every eighth note in the measure (the quarter note beat divided by 2). Example I would also pass the setting for the rule in example II. Example III allows eighth note triplets, but not even eighth notes. Example II would not pass the setting for the rule in example III.

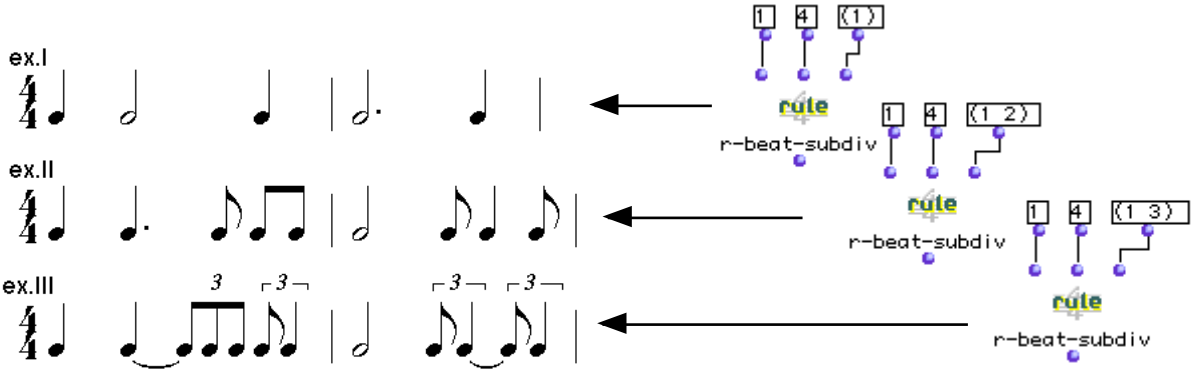



*Fig. 9. The rule r-beat-subdiv*

In tutorial-03c (fig.10) there are time signatures with different beat values in the domain. The rule *r-beat-subdiv* can be expanded to control all different cases. Maximum 5 different beat values can be handled simultaneously by one *r-beat-subdiv* box.

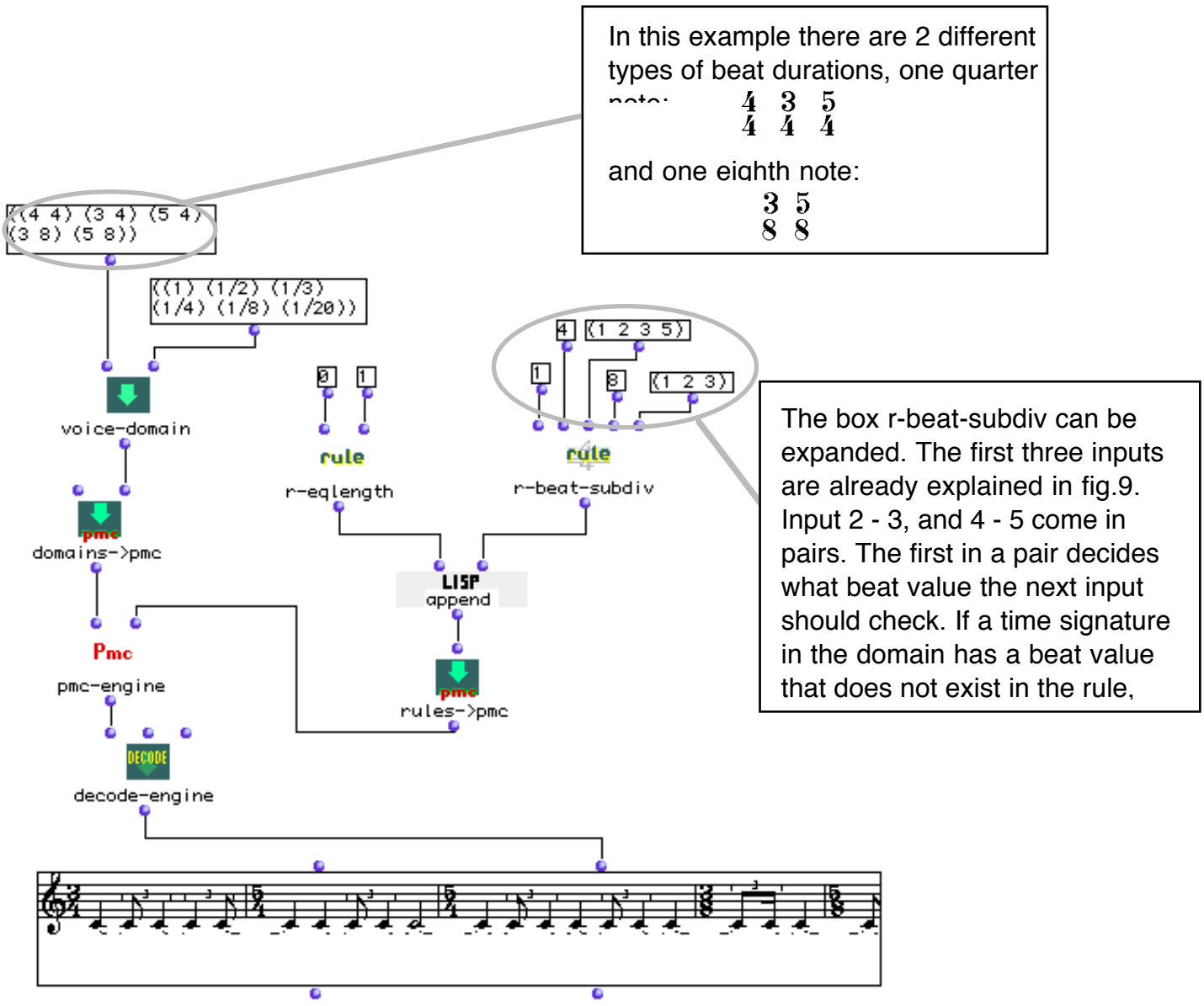



*Fig. 10. Tutorial-03c*

In fig.11 a new rule is added; *r-sync-over-barlines*. It controls where the first event after a bar line can be positioned, i.e. if slurs/syncopations over bar lines are allowed. The first input is the number for the rhythm layer the rule is checking. The second input is a list of possible positions for the first event after a bar line (the following events in the same bar is not affected by this rule). In the example in fig.11 (tutorial-03d), events has to occur on the first beat in all measures. Positions are given as note values (ratios).

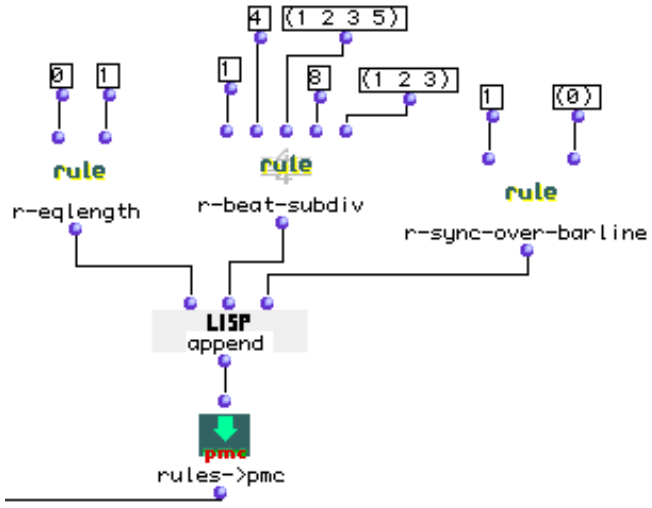



*Fig. 11. Tutorial-03d*

Tutorial-03e does not use single note values in the domain, but rhythm motifs. These motifs are kept intact during the calculation. This example illustrates very clearly the function of the *r-beat-subdiv* rule. Disconnect the rule, evaluate the patch again, and compare the results.

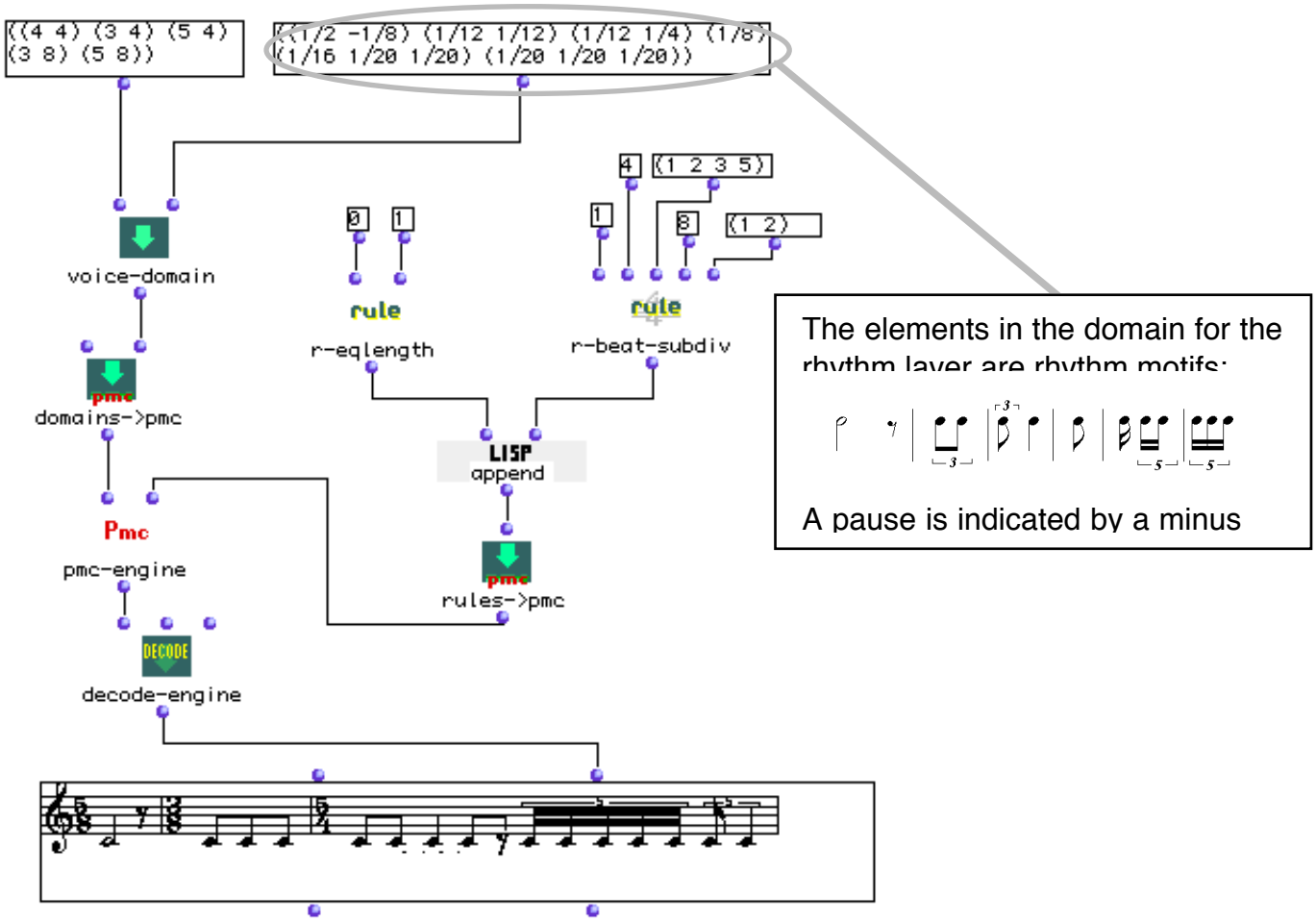



*Fig. 12. Tutorial-03e*

# 4. Hierarchy

So far metrical hierarchy has been discussed. However, other structural hierarchies can exist in music, for example harmonic rhythm, or the position of accents, etc. A hierarchical structure can be visualized by using two or more layers, where one layer only contains the more important event in another layer.

The arrows in fig.13 show the hierarchical connection between the two layers. The start times for the events in the first layer are also start times for some of the events in the second layer. These events are more important than the other events in layer 2, according to the hierarchy. It is up to the composer to interpret this in his music. There are many ways this could be done. The most “classical” one might be to let the first layer represent the harmonic rhythm, and the second layer the melodic rhythm. It is also possible to let the first layer only be a step towards finding the final rhythm, not forcing it to be audible in the music. With some experimentation and experience, the user will find that upper layers in hierarchies can be helpful tools when trying to find a rhythm sequence with certain characteristics.

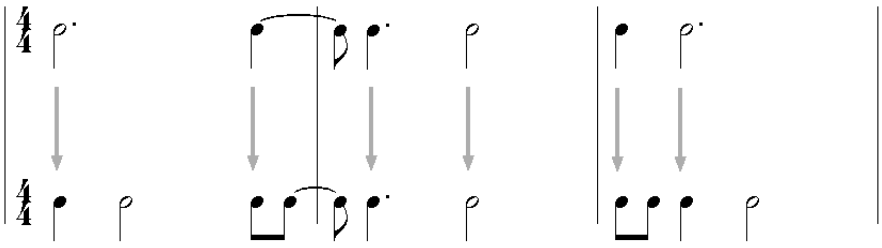


*Fig. 13. A hierarchical structure, visualized on 2 layers.*

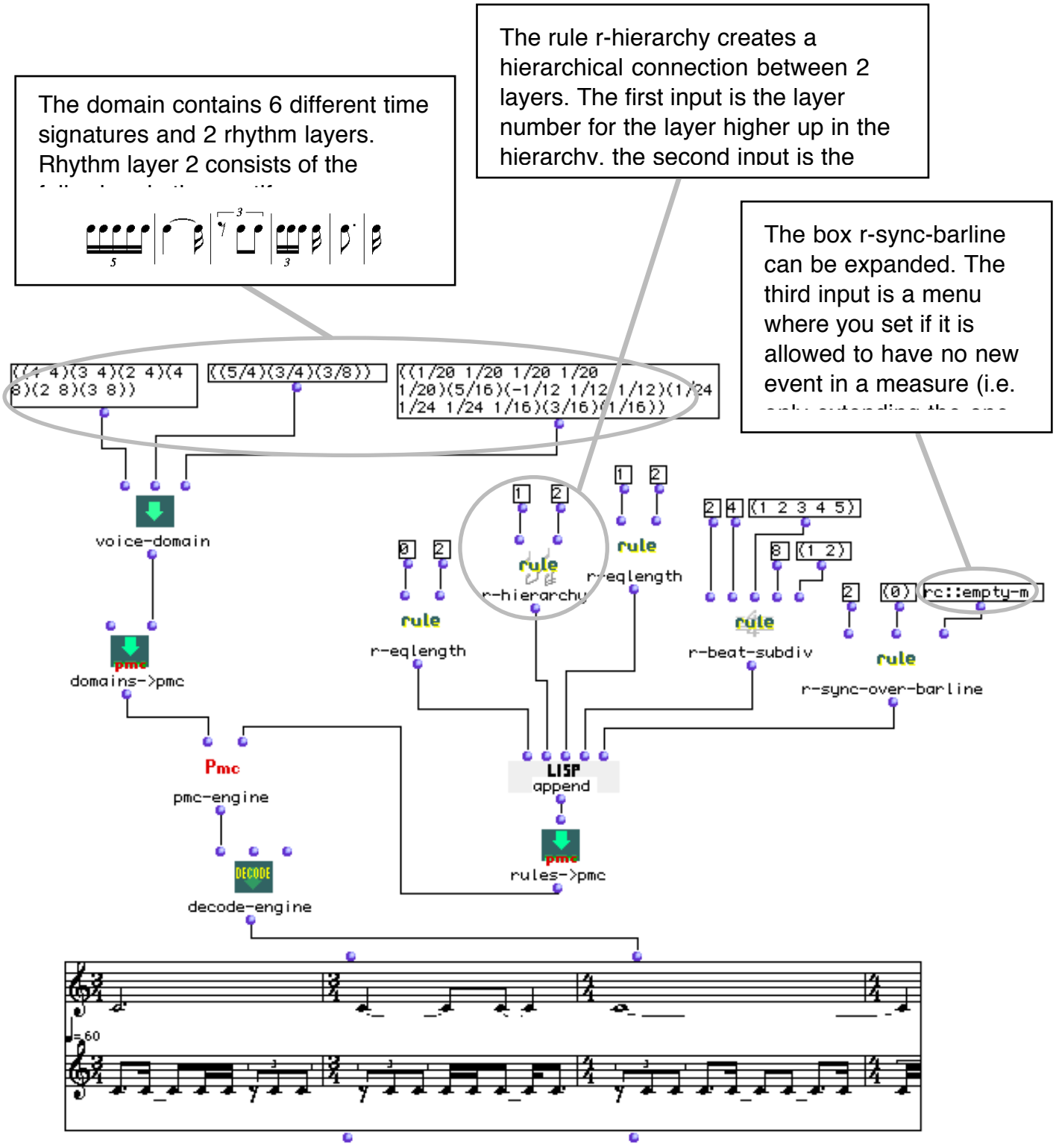



*Fig. 14. Tutorial-04a*

The arrows in fig.15 show the hierarchical connection between the 2 layers in the answer from the example in fig.14.

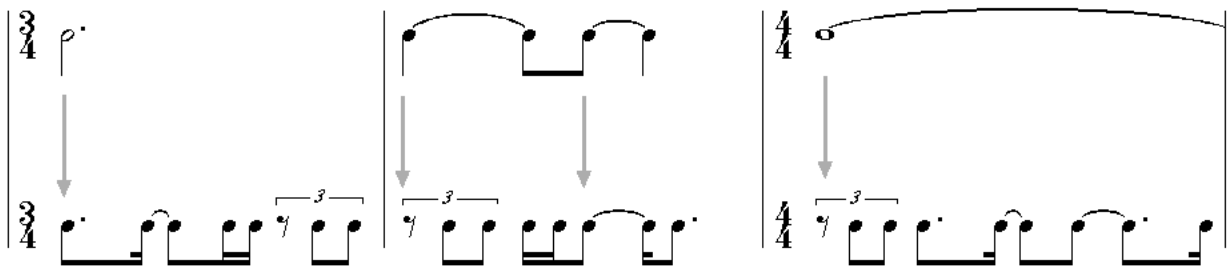


*Fig. 15. The result from fig.14, with the hierarchy marked with arrows.*

The user can also predefine a layer. A predefined layer can not be changed by the search engine. Instead the layer is pre-composed. All rules can refer to a predefined layer in the same way as to a layer that is built of elements from the domain.

To predefine a layer, connect the *preset-layer* function to the input on the *voice-domain* box for the layer that should be predefined. The domain will automatically be replaced by the predefined layer.

The rule r-hierarchy in the example in fig.16, tutorial-04b, makes a hierarchic connection between the predefined layer (which is layer 1) and layer 2. The rule works in the same way as before, but the search engine can not change the predefined layer 1, only layer 2. You can see in the solution in the example that the first layer is identical to the predefined layer connected to the second input on the *voice-domain* box (you only see the start of the sequence).

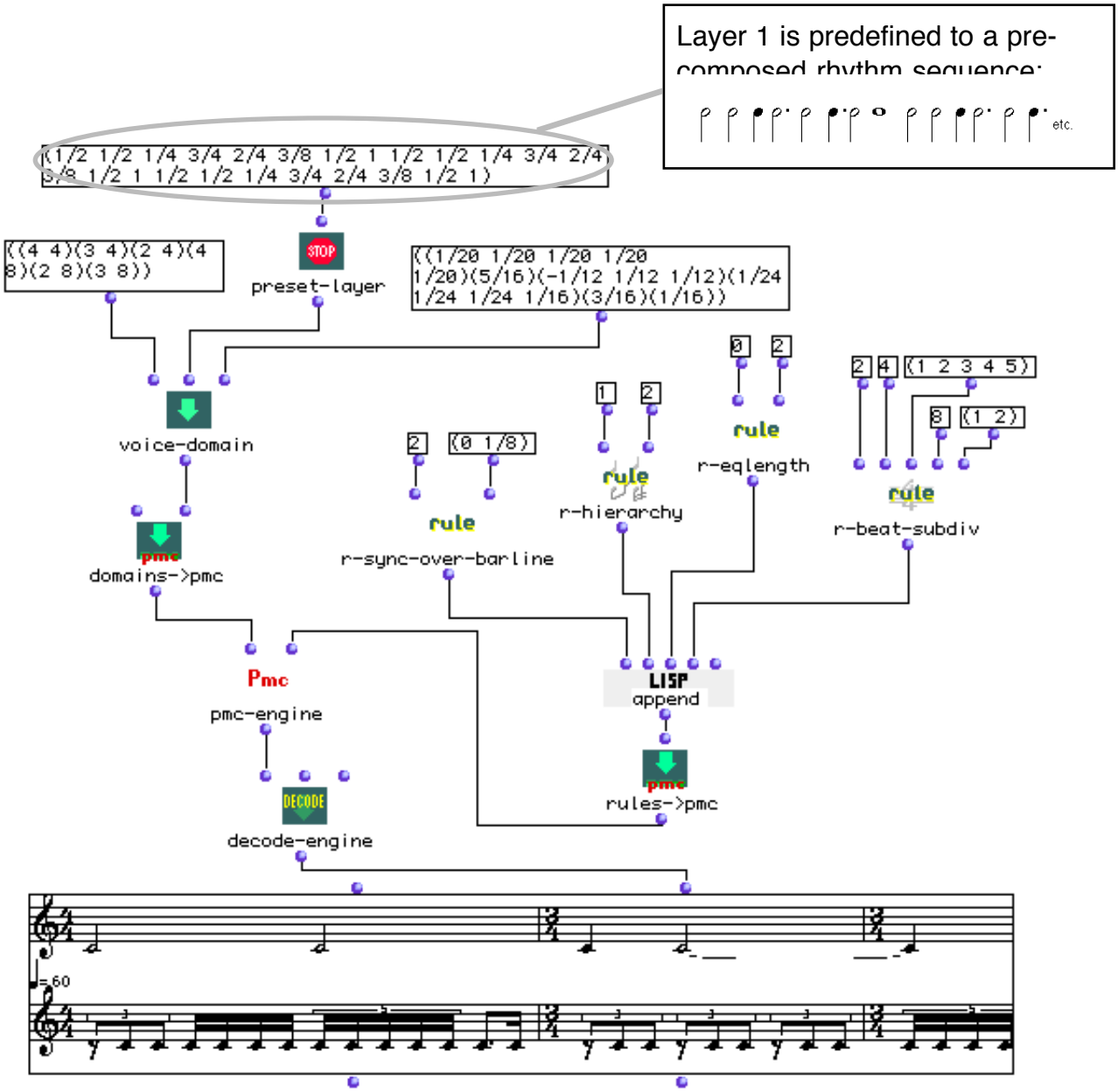



*Fig. 16. Tutorial-04b*

It is possible to view a predefined layer (or several predefined layers) in a score object. The function *view-presets* works similar to decode-engine. However, it should not be connected to the search engine (it accesses the data internally). The function does not affect the solution the search engine finds; it is only used to view all preset layers. The second input is the tempo used by the score object, the third input is a menu where the user can choose the format for the output. If "voices" is chosen, the output can be sent directly to a *poly-object*. The first input should be 0 for now.

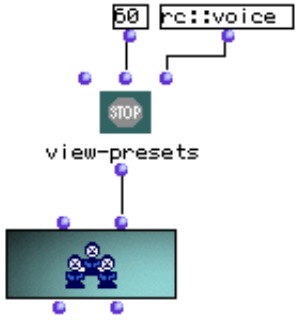



*Fig. 17. View a predefined layer.*

Time signatures can be predefined in the same way as rhythm layers .The function *preset-timesign* will take a sequence of time signature as input. It should be connected to the first input on the *voice-domain* box (the input for the set of time signatures). The sequence of time signatures will be kept unchanged in the solution.

When using *preset-timesign* some additional data has to be added: because of the internal data structure of the library (which will not be explained here), the information that is given to the rule *r-beat-subdiv* must also be given to the *preset-timesign*. In the example in fig.18, input 2 - 5 on both boxes should be identical. If not, the rule *r-beat-subdiv* will not work properly.

If not enough time signatures is given in the predefined sequence to notate the rhythm sequence, the last one will be repeated for some time.

Study tutorial-04c (fig.18). Every time you evaluate the patch the sequence of time signatures will be the same: 4/4  3/8  3/4  4/4  etc.

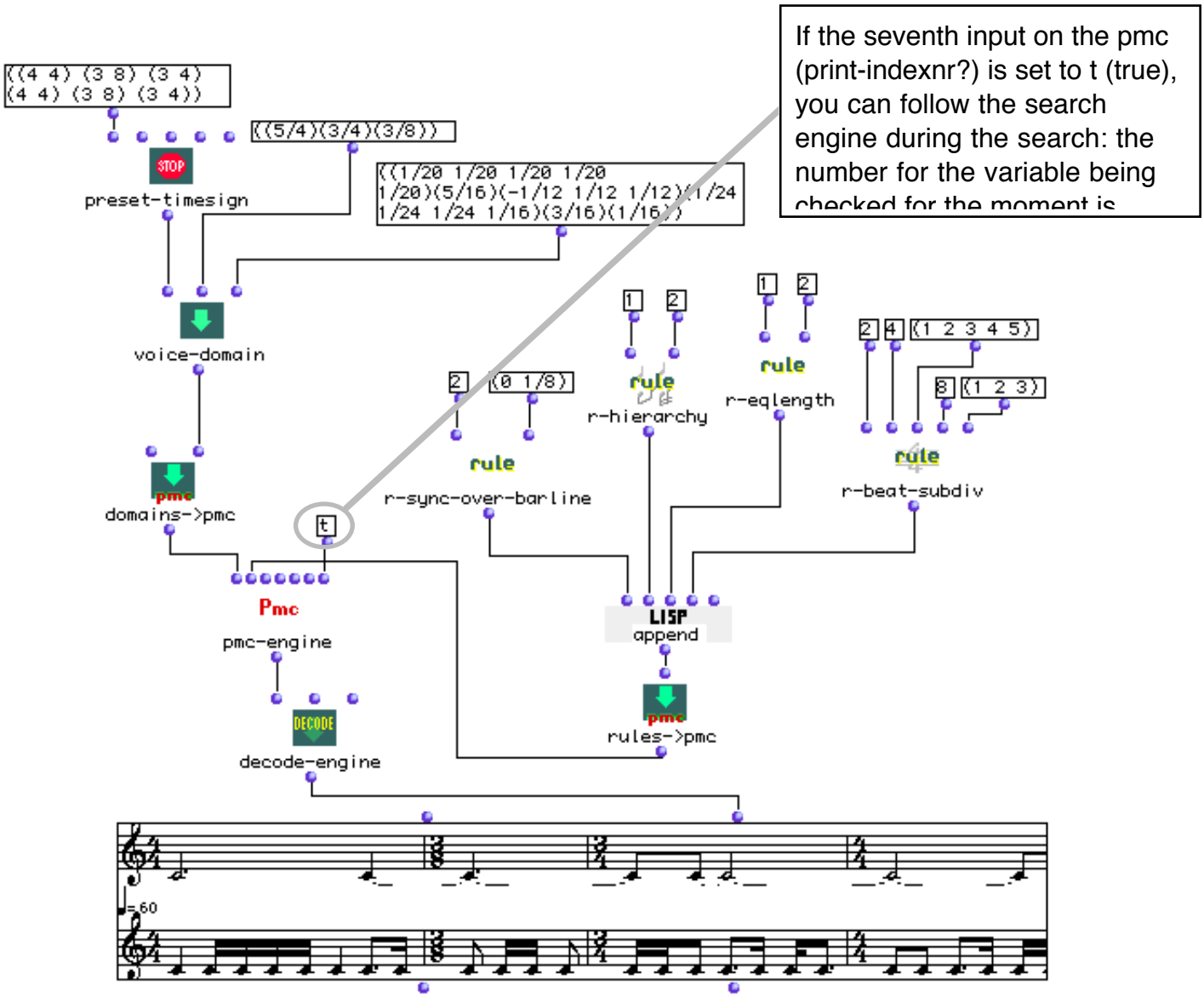



*Fig. 18. Tutorial-04c*

# 5. User defined rules

An important feature of the library is that the user can design his own rules. A user defined rule is always done within a sub-patch in the lambda state. To put a subpatch in the lambda state, select it, press b (a “x” will appear in the top left corner on the icon), and click twice on the “x”. The x has now changed to the lambda sign (λ). See further in the OpenMusic manual.

The subpatch should always have two inputs (which give the search engine access to the rule) and one output. The left input is the index for the variable, an information the search engine uses internal while searching for a solution (called indexx in the input to the user rule tools below). The right input is the current search variable itself (called x in the input to the user rule tools). A rule tests a variable and gives the result true or false back to the search engine.

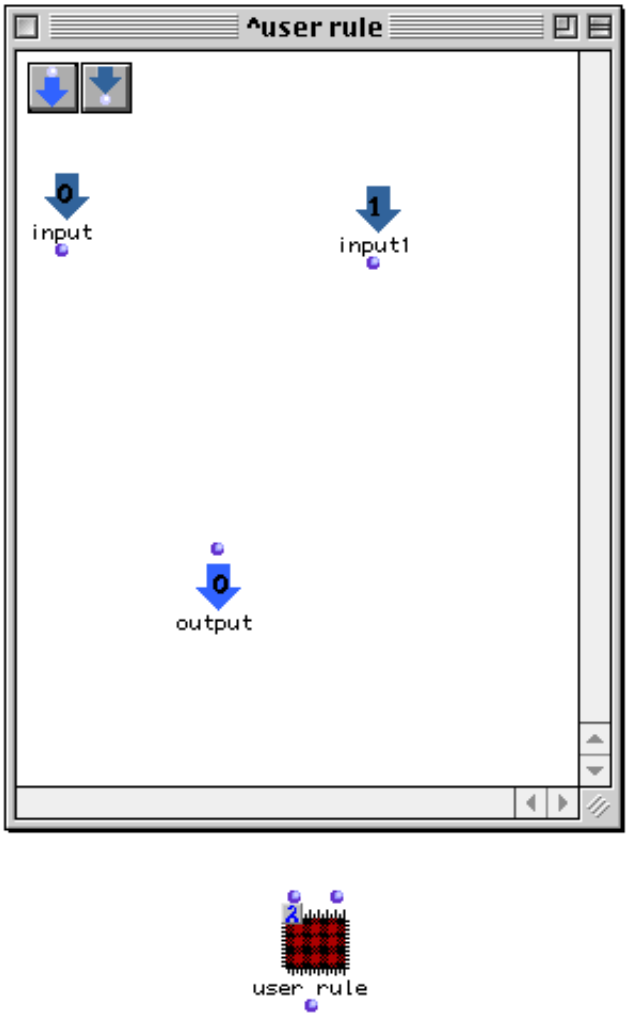



*Fig.19 A subpatch for creating a user defined rule.*

The inputs to a user rule are normally not accessed directly. Instead, there are several tools (in the "user rule tools" menu) that will give the user access to information about the variable, and what the search engine is doing at the moment. A basic example is the function *get-this-cell*:

*Get-this-cell* only has 1 input, and it should always be connected to the second input to the sub-patch. It outputs the current variable (i.e. the element from the domain that is being checked at the moment).

*Get-last-cell* has 2 inputs. They should be connected to the 2 inputs to the sub-patch. It outputs the variable in the same layer as the one being checked at the moment, immediately before the one being checked. The engine has thus already accepted this variable as true according to all rules, but might now reconsider that decision.

*Get-layer-number* outputs the layer number for the variable being checked.

The rule in fig.20 checks if the current variable belongs to layer 2 or not. If not, it always gives true as an output (i.e. it will not affect other layers than number 2). If it does belong to layer 2, it checks if the element is identical to the element chosen (in layer 2) immediately before it. If it is identical, the output is false, otherwise true. This rule will not allow repetitions of elements in layer 2.

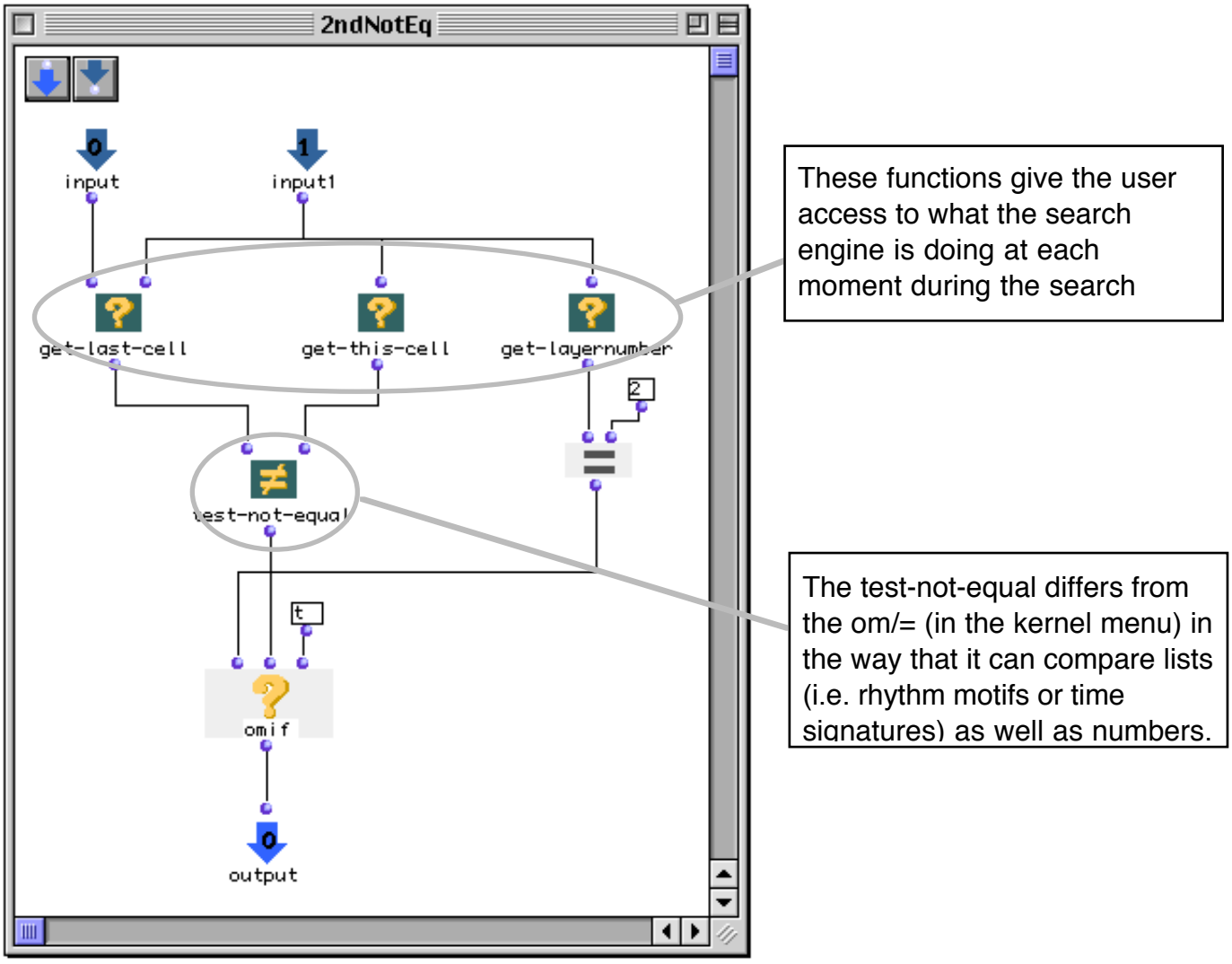



*Fig. 20. The user rule in tutorial-05a (see fig.21)*

The user rules work in the same way as the standard rules. The format requires that the patch for the user rule first is connected to a list object before collected together with other rules (all rules are sent as a list to the *rules->pmc* object, but the standard rules already take care of this). In the solution in fig.21, you can see the effect of the user rule in fig.20: the same duration is never immediately repeated in layer 2.

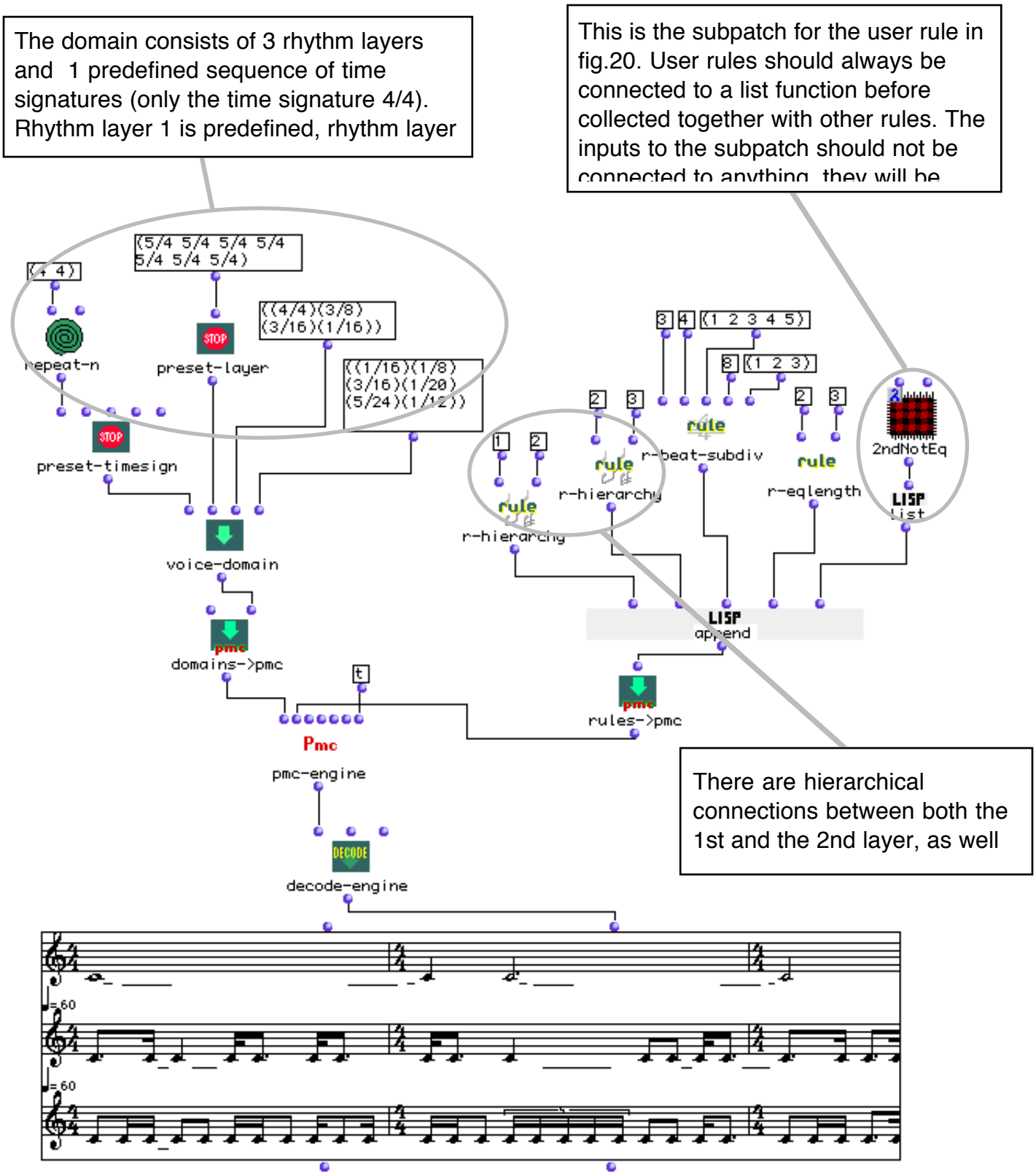



*Fig. 21. Tutorial-05a*

In tutorial-05b (see fig.22), the domain includes 4 options for time signature. This will also make another *r-eqlength* rule necessary, since we have to control the length of the sequence of time signatures.

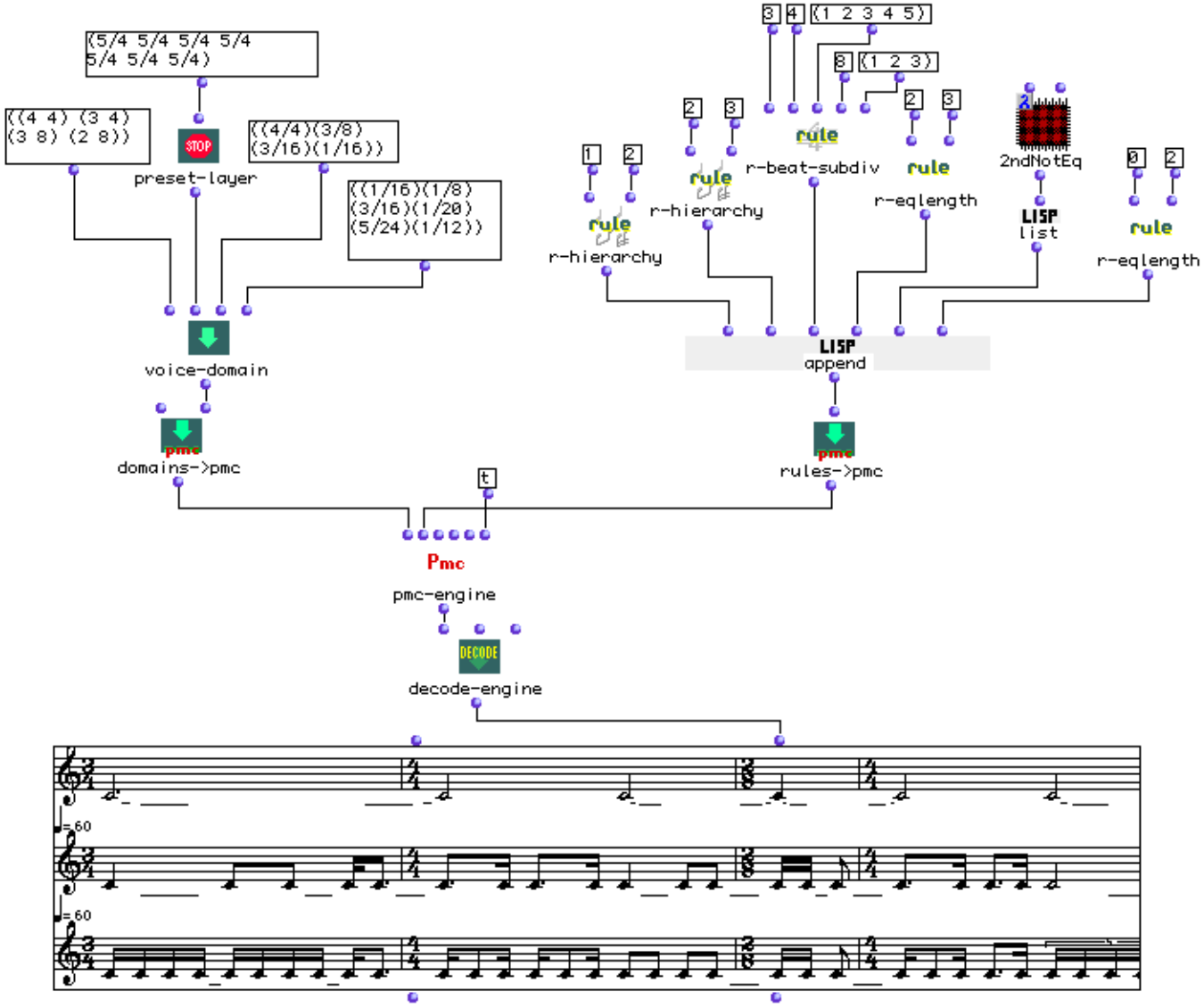



*Fig. 22. Tutorial-05b*

Now a new problem comes in the foreground: because of the complexity of the given problem, the search for a solution sometimes take a long time to run. If you are not lucky, you can get stuck for several minutes or more. Why?

In fig.23 the relations between the rules and the layers are visualized. Layer 2 depends on the events in the pre-defined layer 1. Layer 3 depends on the events in layer 2, and indirect on the events in layer 1. The time signatures depend on the events in layer 3, and indirect on the events in layer 2 and 1.

We give the search engine the instruction to keep all layers of as equal length as possible (we use two r-eqlength to do that). However, we do not know what layer the search engine will start with. Lets assume the first element is picked for the time signatures. There are yet no events in layer 3, so any time signature will do. We assume that the next element is picked for layer 2. It has to fit in the hierarchy to the pre-defined layer. Since there are still no events in layer 3, this layer will not affect the choice. The third element has now to be for layer 3 (it is the shortest since it is the only one that has no events so far). This element has to fit both in the hierarchy to layer 2, and to the chosen time signature. If this is not possible, it will change the element picked before it: layer 2. However, maybe the problem is in the time signature. The search engine will not try that option until all possibilities to change layer 2 has been tried. This is time consuming.

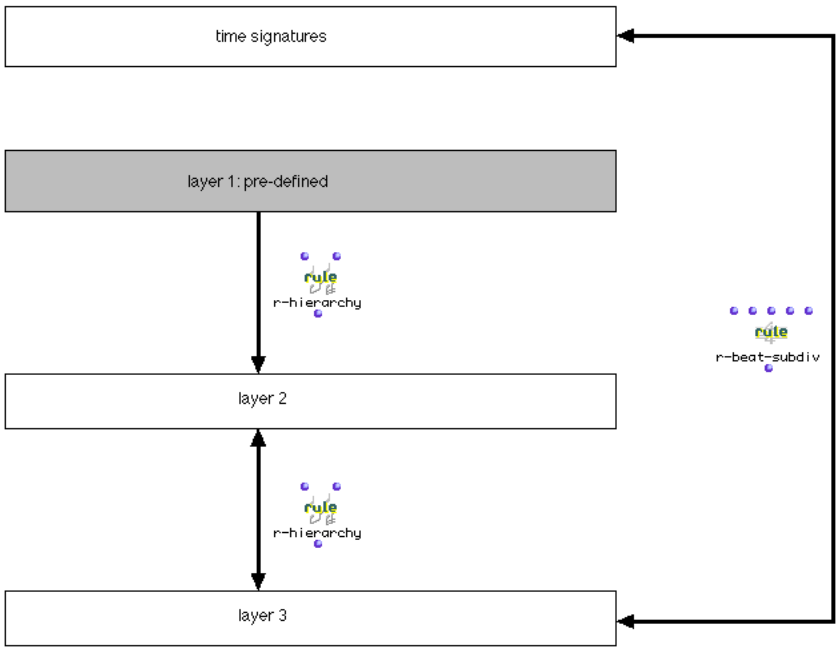



*Fig. 23. The relation between the rules and the layers in tutorial-05b.*

Much harder cases than the one above is possible. To avoid this, it might be a good idea to first calculate layer 2, and then layer 3 and the time signatures in parallel (other strategies are also possible). The rule *r-layerorder* can replace the rule *r-eqlength*. Instead of trying to keep the same length for the 2 layers, it will complete one layer before another layer up to a given time point. After this time point it works in the same way as *r-eqlength*. It is recommended to experiment with these rules when the calculation speed gets very slow. Strategies on how to find a good solution fast is not only a problem in the RC library, but a common problem for all rule based systems. The search process can always be interrupted (commando-. when the listener window is on the top), and restarted, if you get stuck. The engine might by chance find a faster track the next evaluation of the patch.

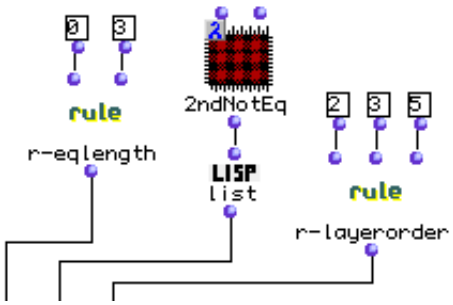



*Fig. 24. An extract from tutorial-05b with the rule r-layerorder replacing one r-eqlength rule.*

In the menu “user rule tools” you will find more tools that can be helpful to build your own rules from your own ideas.

# 6. Heuristic rules

A second type of rules are the heuristic rules. They can be thought of more as “wishes”, than rules that are either true or false. Heuristic rules only exist as user defined. Their construction is similar to regular rules, except for their output: instead of either being true or false they output weights. The search engine compare all possible answers and choose the one with the highest weight. Heuristic rules do therefore not disqualify any answer, they only “prefer” certain answers.

All heuristic rules should be collected in a list and then input to the *heuristicrules->pmc* box, before input to the *pmc*. The *pmc* has to be expanded. The third input is for heuristic rules.

In fig.25 there are 2 heuristic rules, each one in a sub-patch in the lambda state.

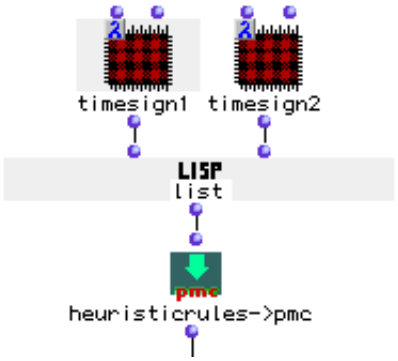



*Fig. 25. The interface for heuristic rules.*

The basic structure of a heuristic rule is identical to the structure for a regular rule. There are always 2 inputs (whether they are used or not), and one output. The inputs access the same data from the search engine as the inputs on a regular rule: index and current variable (see under “User defined rules” above). All tools in the menu “user rules tools” can be used in a heuristic rule. In the example in fig.26, 2 tools are used; *timesign*? and *get-this-cell*.

*Timesign?* should be connected to the second input to the sub-patch. It checks if the current variable is a time signature, and outputs true if that is the case, otherwise false. *Get-this-cell* is already explained above.

The heuristic rule in fig.26 checks if the variable is a time signature or not. If it is a time signature, it outputs the first number in the time signature (i.e. the number of beats in the measure), if it is not a time signature it outputs 0. Since the search engine prefers high weights from a heuristic rule, the heuristic rule in fig.26 will prefer time signatures with many beats.

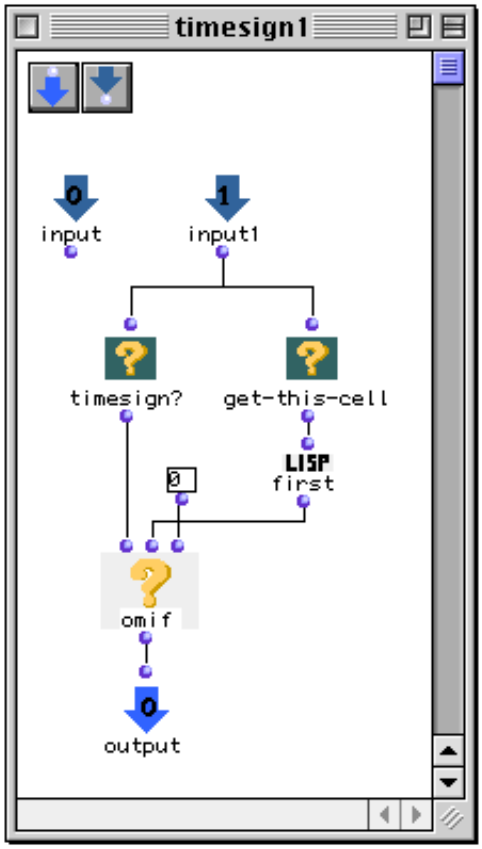



*Fig. 26. A heuristic rule.*

The next example, tutorial-06, is a very practical one. A rhythm sequence of exact note values is given (se fig.27). Only 1/16, 5/16, 5/8 and 1/20 (= one sixteenth note quintuplet) exist. When trying to notate the rhythm using only 4/4 as time signature, we discover a problem: because of a sixteenth note offset at the start of the second measure, the rest of the measure has a very complicated notation.

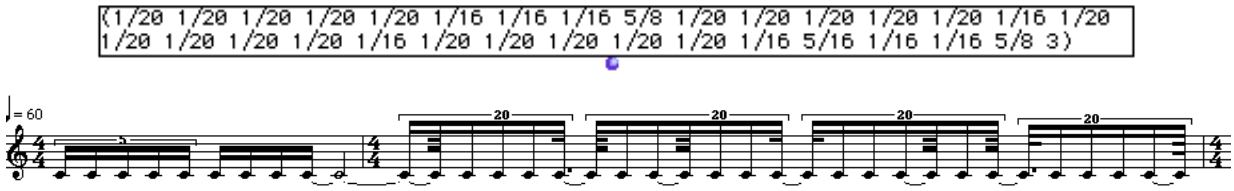



*Fig. 27. A rhythm represented with exact duration values (ratios), and the same rhythm notated with the time signature 4/4.*

In the example in fig.28 the search engine will put time signatures on the above rhythm sequence. The goal is to get an easy-to-read notation (that makes more musical sense). The domain of time signatures includes three different beat lengths: quarter note, eighth note and sixteenth note. The rule *r-beat-subdiv* restricts a quarter note beat to be subdivided up to maximum sixteenth note six-tuplets (the subdivision has to be on-beat, it can not start on a syncope). A eighth note beat can maximum be subdivided in sixteenth note triplets (one eighth note divided in 3 parts), and a sixteenth note beat can not be subdivided.

There are 2 heuristic rules. The first one is the one in fig.26 above (explained in the text above). The second heuristic rule (called *timesign2*) looks at the beat duration, and it prefers quarter note as the beat value in front of eighth note beat value, in front of sixteenth note beat value. Double click on the sub-patch to study how the rule works.

The 2 score objects in fig.28 contains the same rhythm. The bottom one has a much more understandable notation, and makes more sense than the top one. Observe that the search engine has not changed the rhythm itself, since the rhythm is a pre-defined layer.

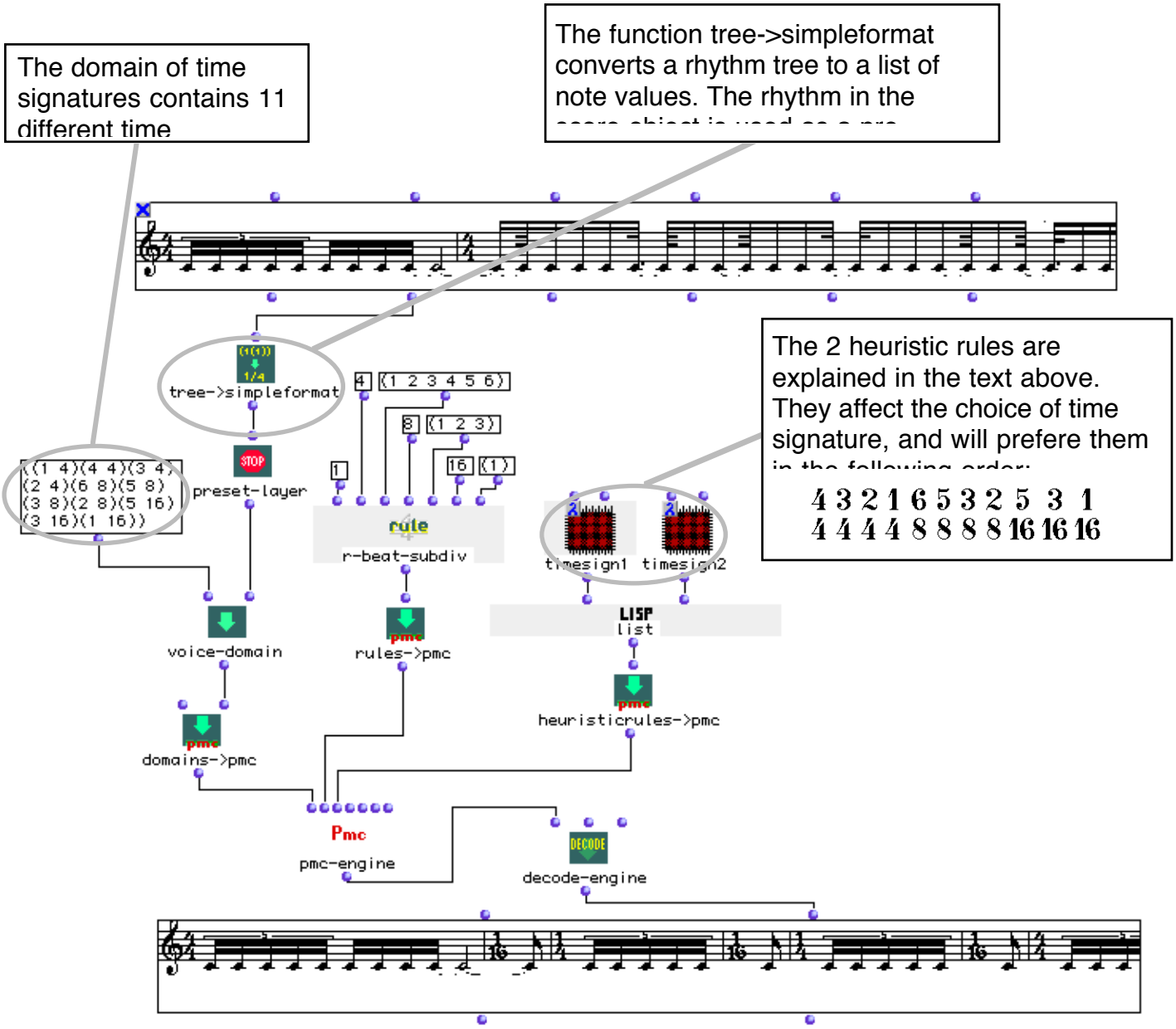



*Fig. 28. Tutorial-06*

# 7. Lock sections of a solution

Every time you evaluate a patch in the RC library, the *pmc* will find new solution, which might be different every new evaluation. If you get an answer you don't like for some reason, it is often enough just to evaluate the patch again to get another version. However, a much more likely situation is that you like some sections of one solution, but not other sections. Up until now, the only way to find an alternative solution has been to destroy the old solution.

The function *store-section* help you avoid this. It will store a section of the last found solution (= the last solution any *pmc* connected to a RC rules interface found in any window in OpenMusic). You can generate instances of the class *stored-section* to store the information about a section of a solution for future use.

The object stored-section (and the output from the function store-section) can be decoded by the function *decode-stored-section*. It works similar to *decode-engine* and *view-presets*. *Decode-stored-section* can be expanded, and the second input will let you choose the output format in the same way as on the other functions mentioned. Default it will output a poly object.

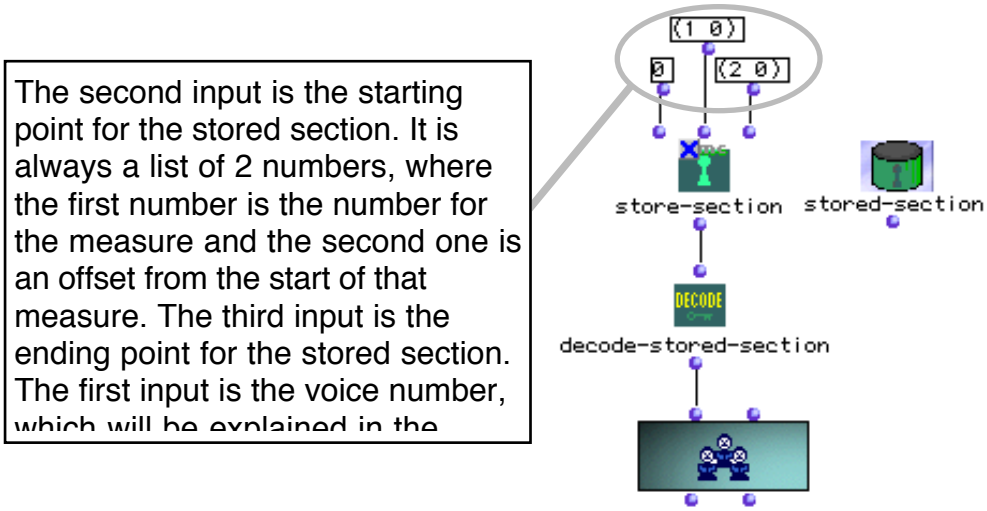



*Fig. 29 The interface for storing a section.*

The user can use the information from the stored section to force the engine to keep a section in the new solution identical to the stored section. You can lock all information in the section to the stored information (i.e. all rhythm events and time signatures), or only lock one or several layers in the section (or only keep the time signatures).

The rule *r-lock-section* will lock a section in the solution to a stored-section. As first input it takes a stored section from an earlier calculation (either directly from the output on the *store-section*, or from a *stored-section* object). On input 2 - 6 (the box can be expanded) the user defines which layers should be locked. If the input is hidden, or *nil* is given as an input, it will not affect the result.

The rule *r-lock-stored* works as any other rule, and should be collected with all other rules before input to the *rules->pmc* box.

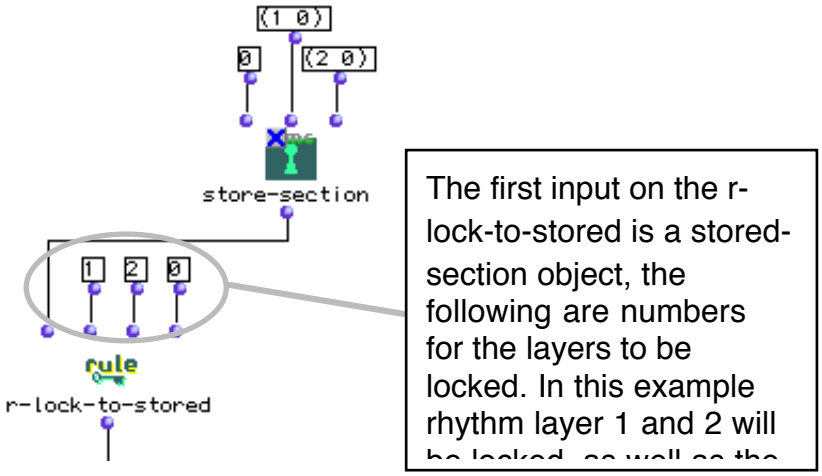



*Fig. 30. The interface for locking a section.*

A section is stored with the information about at what time point it existed originally. When storing the section, you define the time points in *measures and offsets*. This is then converted to a time point expressed in "pure duration". If locking the section to its original point in time when searching for a new solution (which is the default), this might not be the same measure number any more. For example, if the starting point originally was in measure 3, and if the time signatures originally was 4/4, the same time point in the new solution will be measure 5, if the time signature changed to 2/4. After a section is stored, it is the original time point in "pure duration" that is valid, not its measure number.

If the box *r-lock-to-stored* is expanded completely, the seventh input is used to define a new start time for the section in the new solution. A section can thus be moved in time. However, it is easy to move a section in such a way that it is impossible to find a solution anymore. There are still only a limited number of note values for the engine to use, and all rules has to be followed during the search for the locked section (they will be checked *for the whole sequence* by the engine). It is always harder for the engine to build a start for the solution that has to fit a locked section that exist later in the solution, than to start with a locked section (in the latter case, the user gives the solution to the engine, and the engine just confirms it).

The example in fig.31, tutorial-07, is similar to tutorial-05b. The only new rule/setting is the *r-lock-to-stored*. The function *store-section* can not be evaluated before the patch has been evaluated once. A user can be confused to think that the information the *store-section* stores is identical to the notation in the score object showing the solution. This is normally the case, however only if the score object actually views the last solution. The *store-section* access the last solution via internal vectors that are directly linked to the search engine. If the engine was interrupted before the result was output, the information in the vectors might be corrupt, and is very unlikely to be the same as in the score object.

Experiment with changing the stored section in tutorial-07, and changing the settings for the *r-lock-to-stored* rule.

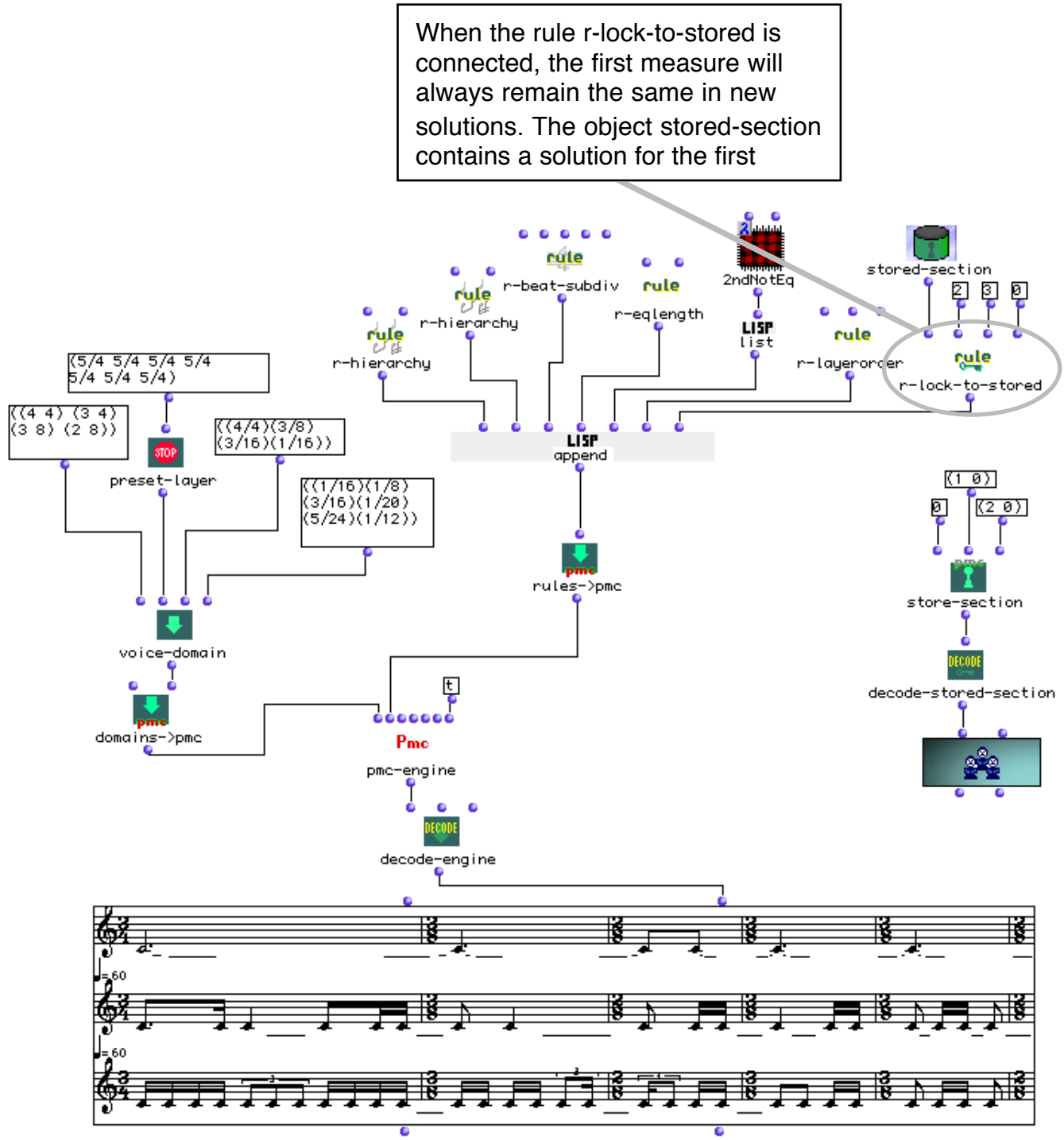



*Fig. 31. Tutorial-07*

# 8. Polyphonic structures

The examples has so far output solutions with several rhythm layers, divided into measures. The RC library supports maximum 4 rhythm layers + time signatures simultaneously. The concept of what a layer represent can be different from case to case. In fig.6 the two layers are two voices in a canon. In fig.14 the two layers make up a hierarchical structure, where the first layer only visualize the hierarchical structure in the second layer.

To make it possible to expand the concept of what layers represent, the RC library supports maximum 7 voices in a musical structure, where every voice has its own hierarchical structure, visualize on up to 4 layers (up until now, all examples have only used layers

within one voice). Each single voice can even have separate time signatures, so that each musicians play in his own independent meter. Maximum 7 voices, which each one has maximum 4 rhythm layers and independent time signatures, can simultaneously exist in a calculation. This equals 35 layers, which is probably more than calculation speed will allow the user to work with - the complexity that soon will be reached will make the time to find a solution very long.

The functions *domains->pmc* and *rules->pmc* can be expanded. Each new input represent a new voice. Each voice has its own, separate domain of note values/rhythm motifs/time signature. Only rules connected to the corresponding input on the *rules->pmc* will affect a voice. You can see the structure for the interface in fig.32. The example has three voices with independent domains and rules for each voice.

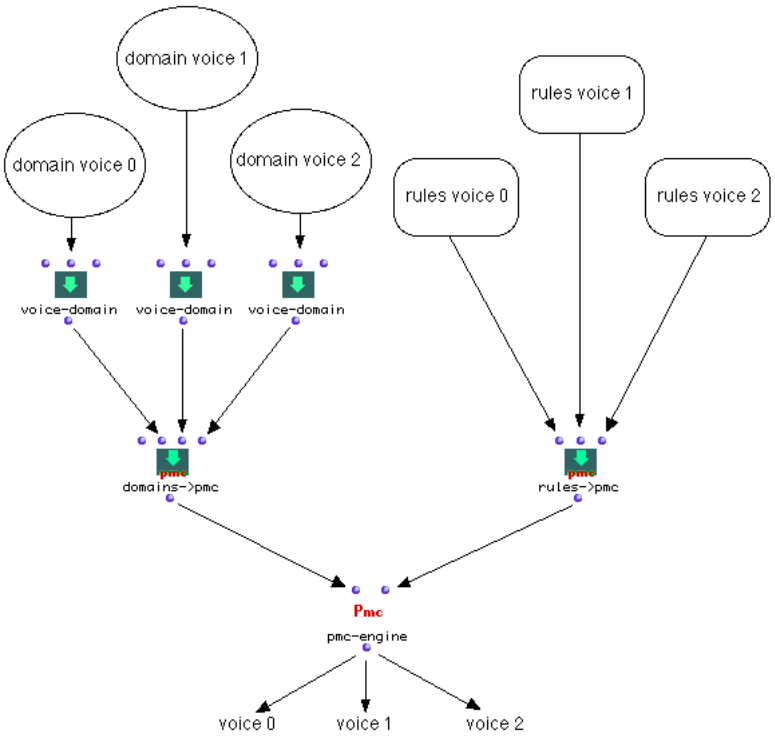



*Fig. 32. The structure for a patch with three independant voices.*

There are two type of rules: regular rules within one voice (this is the type that has been used up until now in all examples), and global rules. A global rule is a rule between two voices. It can be identified by that its name starts with a “gr-”, while the name for regular rules always start with a “r-”.

A global rule has to be connected to the inputs for both the 2 voices it should check on the *rules->pmc* (there is one exception; the rule *gr-canon*). If not, it will not have access to the domains for both voices. In fig.33, there is an example with three rules. The first input on the box *rules->pmc* are for all rules that should check voice number 0, and the second input are for all rules that should check voice number 1. One of the rules is global between the two voices, and connected to both inputs. The other two rules only affect one voice each.

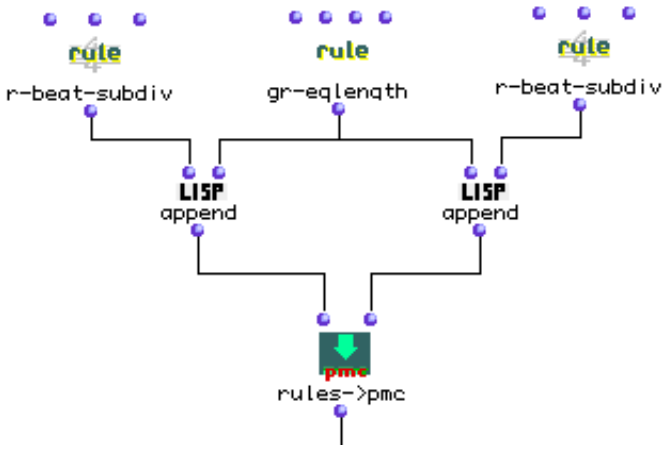



*Fig. 33. Regular and global rules.*

In the following example, tutorial-08 in fig.34, we will make a canon in accents between 2 voices. The rhythm in the voices will not directly be in canon, but the accents in both voices will follow each other in canon.

Study the example in fig.34. The voices have identical domains. That is why only one *voice->domain* box is used; it is connected to both input 2 and 3 on the *domains->pmc* box, so that two copies of the domain will be created (they will actually not be identical, since they will differ in the information to what voice they belong - this information is added by the *domains->pmc* box). There are always the same meter, 4/4, in both voices, and the basic structure, layer 1, is shared by both voices.

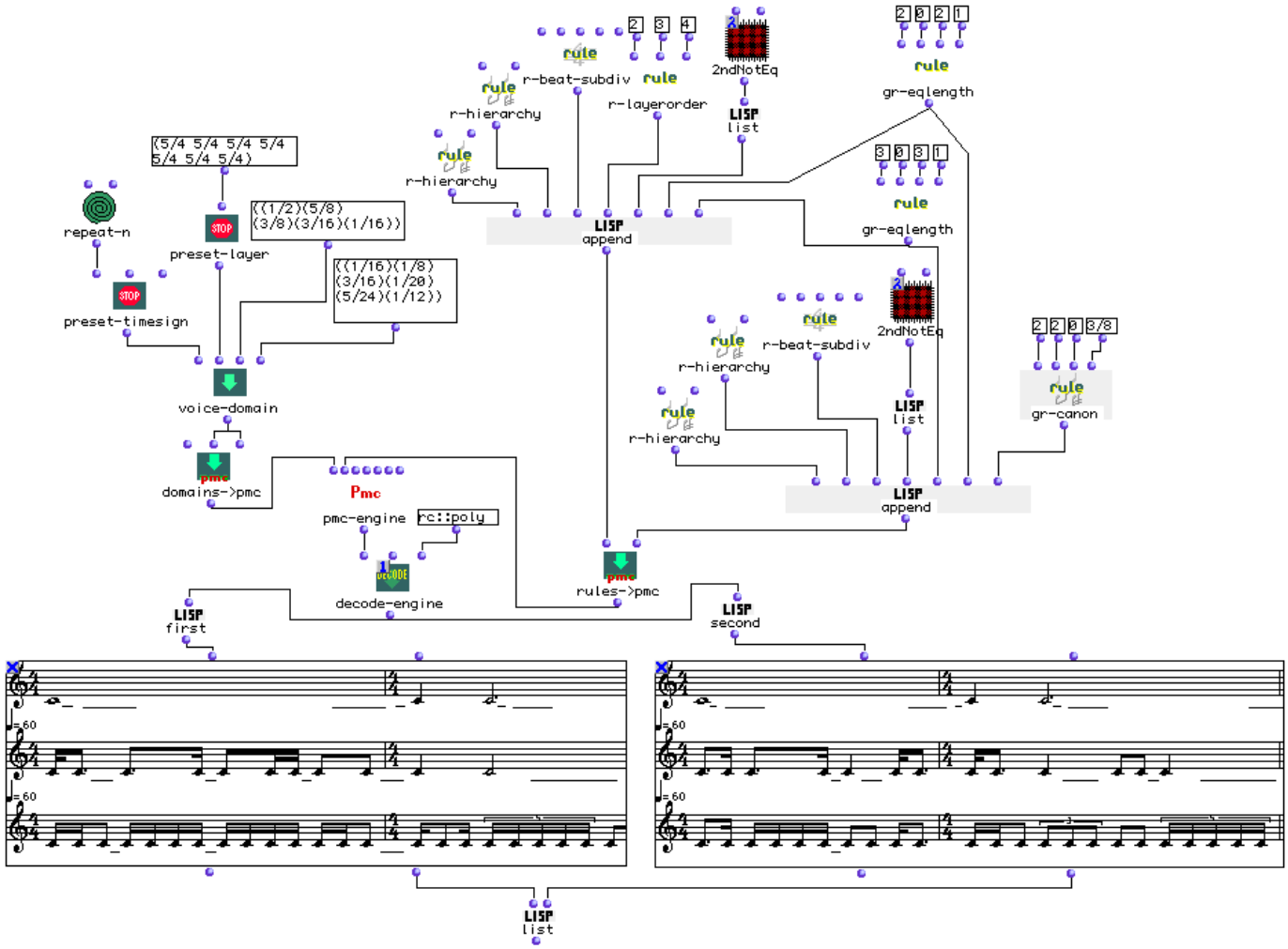



*Fig. 34. Tutorial-08*

The domain and the rules in one voice in this example are similar to the domain and the rules in tutorial-05a (fig.21), but rhythm layer 2 has a slightly different domain. There are hierarchical connections between layer 1 and 2, and between layer 2 and 3. The user defined rule does not allow immediate repetitions of elements in layer 2.

There are global versions of the rules *r-hierarchy*, *r-eqlength*, *r-layerorder* and *r-canon*. The global rules have the prefix *gr*, and are thus named *gr-hierarchy*, *gr-eqlength*, *gr-layerorder* and *gr-canon*. They all work in the same way the regular rules do, but they need extra inputs since the user has to specify to which voices the layers belong.

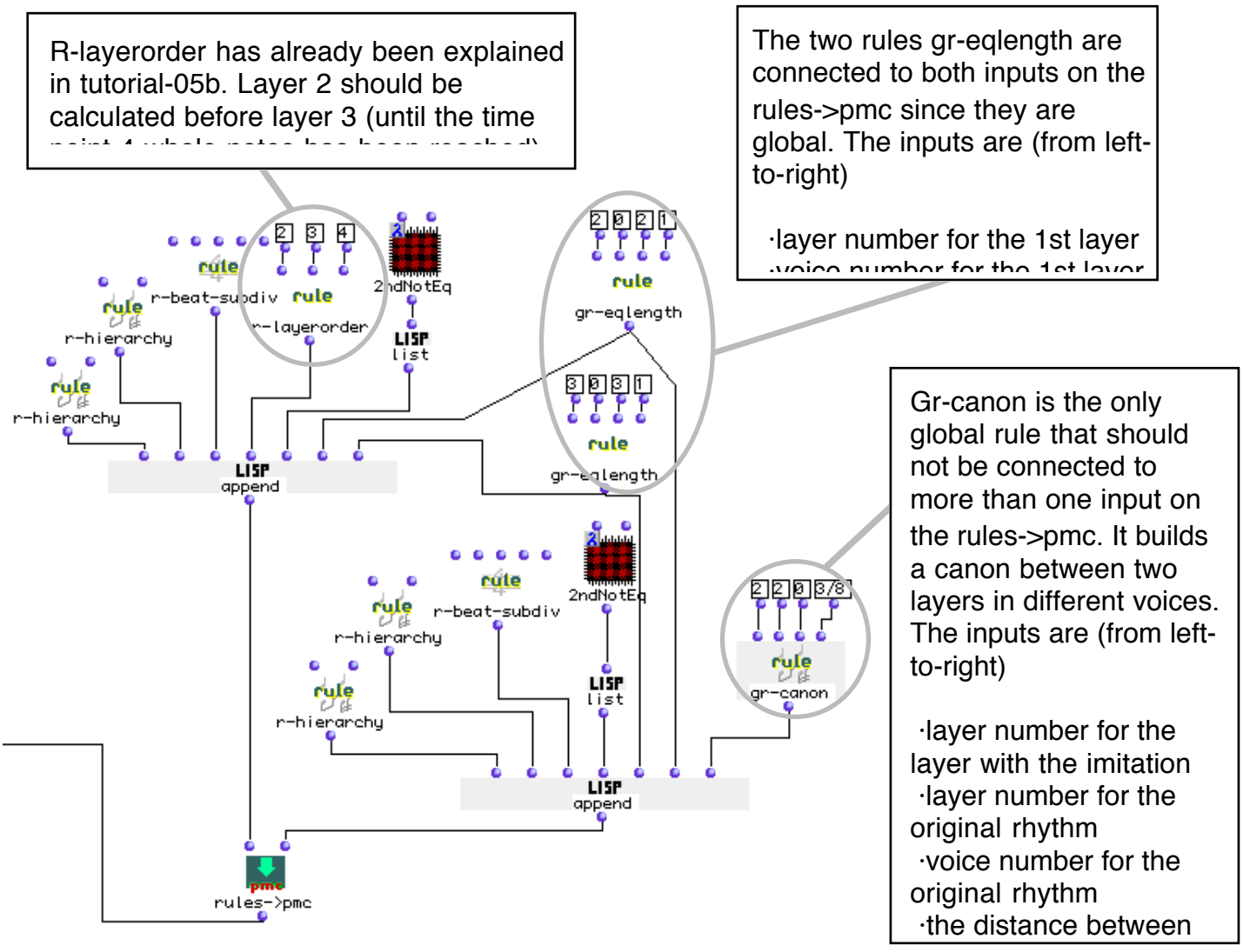



*Fig. 35. The rules in tutorial-08. The settings that are not indicated in the picture are identical to the settings in fig.21.*

The complexity of the problem is now increasing. If you do not plan in what order the layers should be calculated, the search engine will soon get stuck. Of course you will always find a solution without this plan, however you will probably have to wait for a long time, while the search engine first tries a high number of dead ends while searching for the solution. With a plan, you can help the search engine to find a good track to search on.

In fig.36, the relations between the rules and the layers in tutorial-08 is visualized. Layer 2 in both voices depends on the events in the pre-defined layer 1. Layer 2 in one voice also depends on layer 2 in the other voice, since there is the canon rule. Layer 3 in each voice depends on the events in layer 2 in the same voice, and indirect on the events in layer 1 and the events in layer 2 in the other voice. Layer 3 also depends on the time signatures. As you can see in fig.36, everything is linked to each other. If you change a small detail in one layer, you might have to change all other layers.

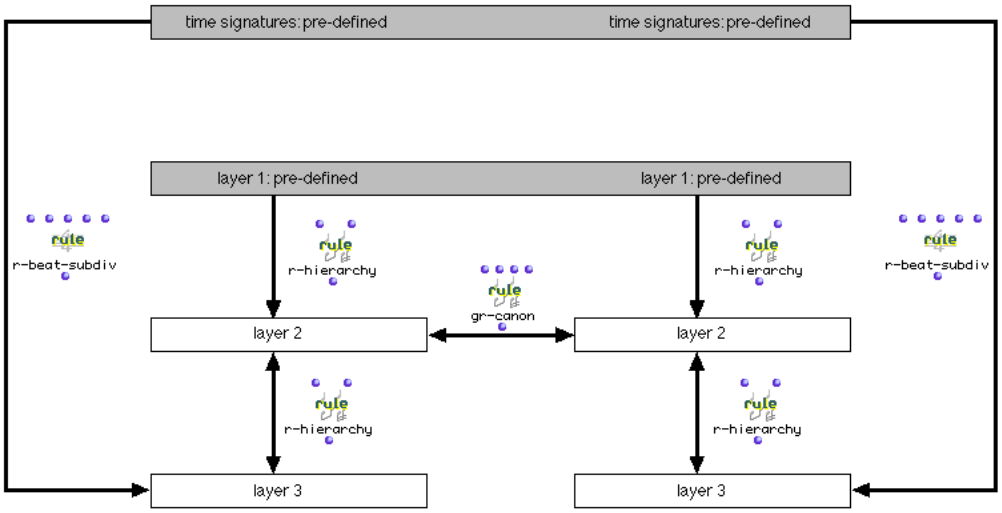



*Fig. 36. The relations between the rules and the layers in tutorial-08.*

The strategy used in tutorial-08 is to first calculate layer 2 in both voices (in parallel), and then layer 3 in both voices (in parallel). The two rules *gr-eqlength* make sure that the two layer 2 are calculated in parallel, as well as the two layer 3. The rule *r-layerorder* only exist in the first voice, but since the two layer 3 have to be calculated in parallel, it will also affect layer 3 in the second voice.

This strategy is a faster than calculating all four layers in parallel (you can easily try that option by changing the *r-layerorder* to *r-eqlength*). It might always be a good idea to try a different strategy if the search engine gets stuck. Sometimes complex relationships between layers can be very hard to grasp.

The solution in fig.34 is clean written in fig.37 below. The to voices are written on top of each other (as in a traditional score). The two layer 2 are marked by accents. The canon is between the rhythm the accents make up. The accents in the second voice starts 3 eight notes later than the accents in the first voice.

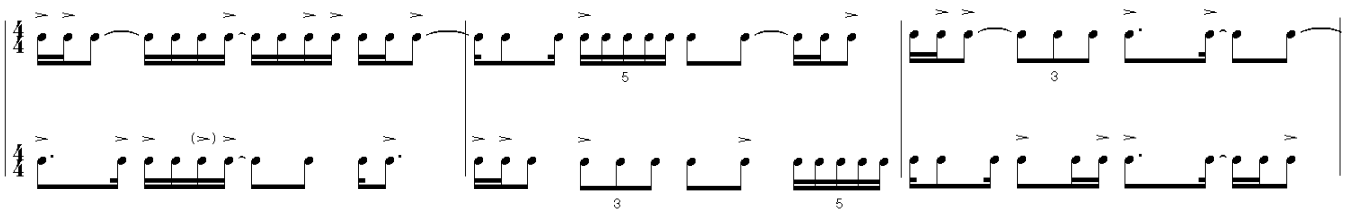


*Fig. 37. A possible solution from tutorial-08 clean written.*

# 9. Csolver compatibility

Normally the *pmc* in the "OMCS" library is used as the search engine for the RC library. However, the *csolver* from the "Situation" library is also supported. Different engines work with data structured in different ways. To be able to use the *csolver*, you have to change four of the functions that format and reads data to and from the *pmc*, to equivalent functions for the *csolver*. These boxes are not present in the menus, but they can be used by command-clicking and dragging the mouse in a patch, and then typing there names (see the OpenMusic manual).

The functions that support the *pmc* and the equivalent function for the *csolver* are:

| | | |
|---|---|---|
| rc::domains->pmc | => | rc::domains->csolver |
| rc::rules->pmc | => | rc::rules->csolver |
| rc::layer-in-solution | => | rc::layer-in-sol/csolv |

The csolver box has first to be expanded with one extra input. This 7th input is the input for the domain (<data>).

The rules should be connected to the 5th input on the csolver (<cnstr>).

The number of variables in the solution is now set on the first input on the csolver.

ALL OTHER INPUTS ON

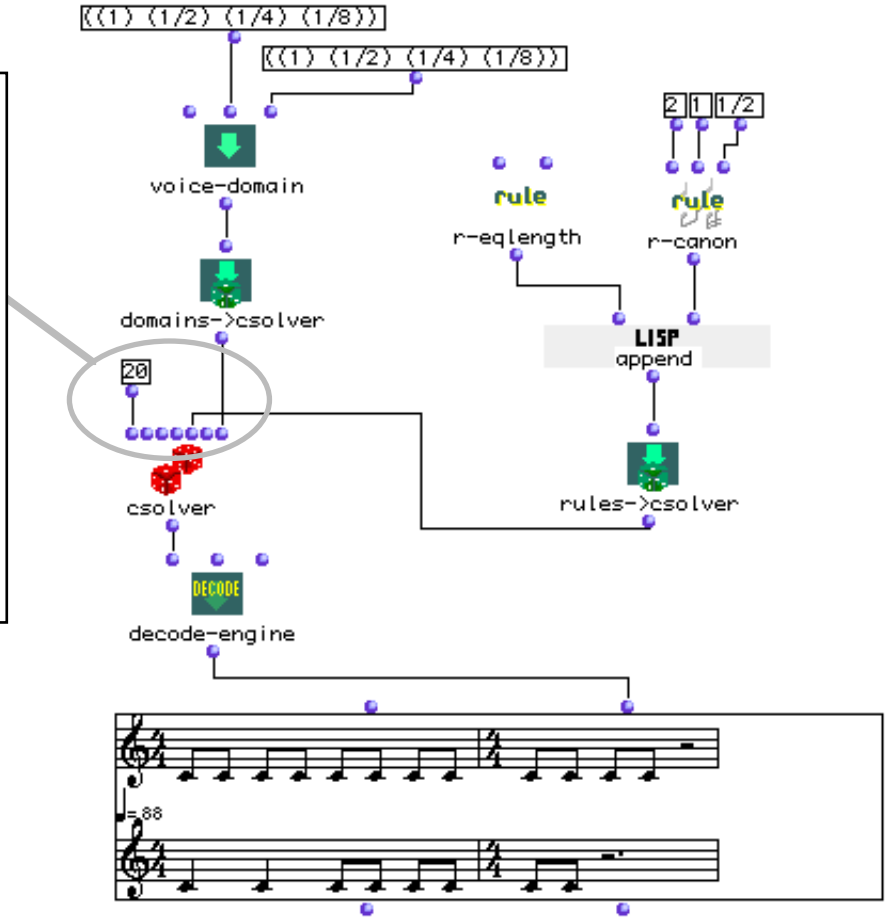



*Fig. 38. An example of using the Csolver as search engine. Compare with tutorial-02b (fig.6).*

The csolver (in the Situation 3.0 library) does not work the same way for example when it comes to reordering the domain randomly. This will not be explained here, but the user will soon discover some differences in behavior.