Logol Designer And Language Tutorial

by

Olivier Sallou - IRISA

History:

v1.0 : Creation

v1.0.1 : Fix errors in Parental comparison chapter.

v1.0.2: Change special characters for end and dist constraints due to encoding issues

v1.1: Add new features (optimal size constraint and alphabet constraint)

v1.2: Add macro controls

v1.3: Add info on morphism

Index

Index des illustrations

Index des tables

1 Starting Up...

1.1 Introduction

Logol is a language used to describe some patterns in a DNA/RNA/Protein sequence. It focuses on biological meaning patterns to help biologists and bio-informaticians formulate some patterns and create an appropriate model. LogolMatch will then scan a sequence with this model to find the expected patterns.

This tutorial will describe this grammar, how to use its language to define patterns, and how to use LogolDesigner to create some grammar models graphically.

1.2 Audience and requirements

The intended audience is people needed a formal model description language for biological patterns and people using LogolMatch.

There is no prerequisiste nor biology or IT knwoledge needed.

As a pattern description language, Logol expects that a pattern can be literally described to be applied on a sequence. The tool does not find a-priori patterns from a set of sequences, it is intended to find a known pattern only, though the pattern can be “soft”, meaning that unknown parts can be introduced, or parts with error ranges.

1.3 Software

LogolDesigner is available at http://webapps.genouest.org/LogolDesigner, on the GenOUEST bioinformatics platform. The grammar can be designed graphically from a web browser (Internet Explorer, Safari, Firefox, Chrome). It expects quite recent version browsers.

Screencasts can be found at this URL for basic usage of the software and quick startup.

2 Language concepts

Besides language writing or graphical editing concerns, it is first required to understand the Logol concepts. Those concepts will be find in both ways of modeling.

Here is the language graph:

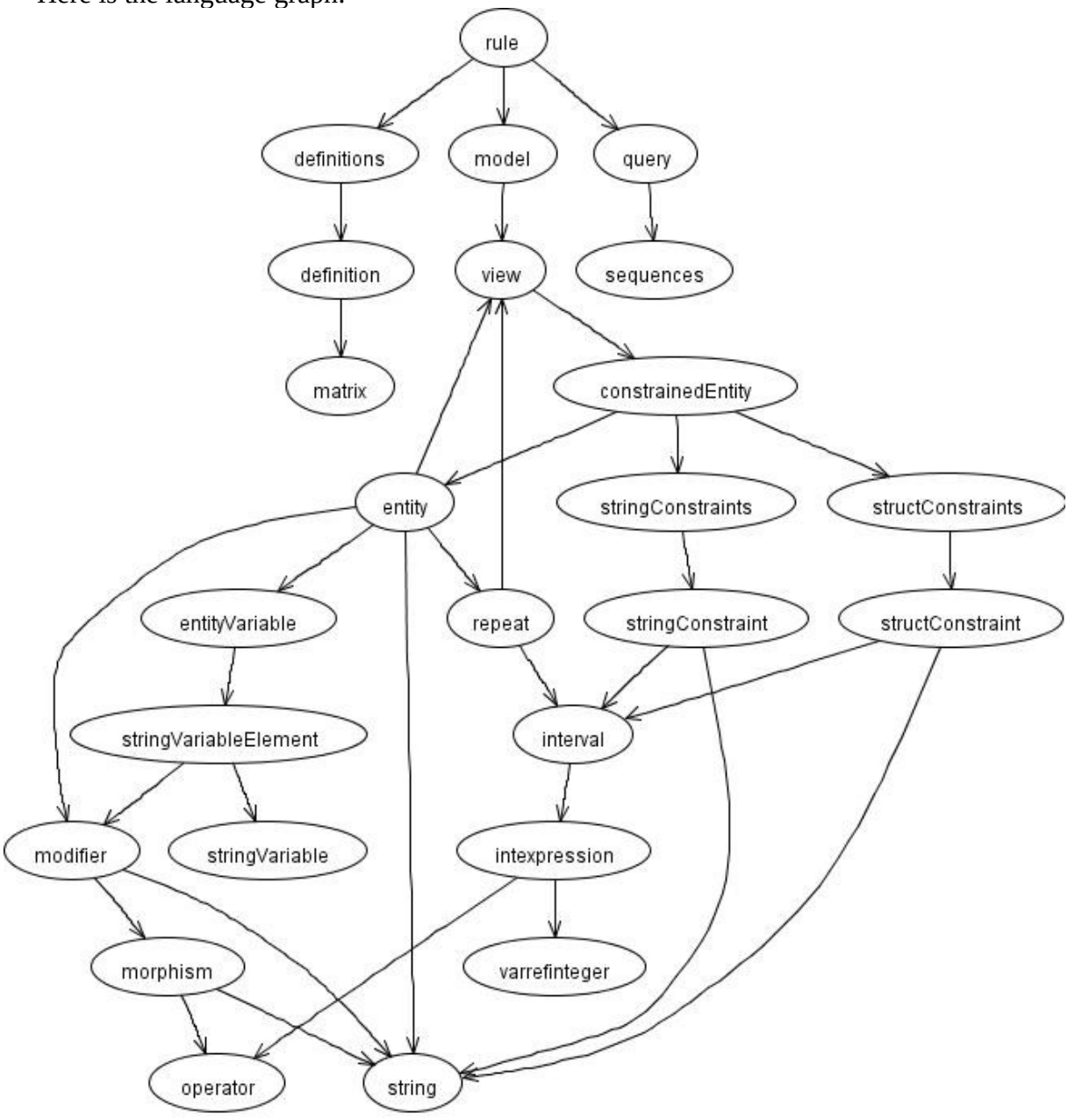

Illustration 1: Grammar graph

Don't be afraid by this graph, you don't need to understand the whole thing. This tutorial will briefly the most important parts of the components. With the graphical interface, most of those relationships are hidden within the interface.

2.1 Rule

The rule is the entry-point in the language (grammar). This is the place where one must define the models to apply on a sequence. Several models can be applied on a same sequence, with the possibility to use variable from one model in the other.

For example, a first model can be used to find an anchor in a sequence, then to apply other models, based on this anchor to find other sub-components.

For the models defined in the rule, the search will not start specifically at position 1 in the sequence. The tool will try to match the first entity of the model anywhere in the sequence. As a consequence, it is important (but not mandatory) to set the first entity as a string anchor, to get an easier starting match. The more the anchor is strong (well defined), the faster will be the search.

2.2 Models

Compared to programmatic languages, models can be compared to functions. Models define a set of words, other models calls and constraints to match a part of a sequence.

For example, a model A could be:

```
ModelA    {
          search “acgt”,
          search “cg”, repeated 3 times
          search model B
          }
with:
model B   {
          search “tgca”
          }
```

Texte 1: Models usage example

2.3 Views

A view can apply some constrains on a group of words or models. It can be used to limit for example the total size of a set of words in a match.

Example:

(search “acgt” with a max distance cost of 1, search “cgta” with a max distance cost of 1) whole max distance cost of 1

The view can be seen here as the parenthesized group.

2.4 Entity

Entities are the search words in the sentence, with text search comparison. Entities can refer each other via variables in constraints (content constraint for example).

Entity references do not imply left-to-right constraints, e.g. a word A can refer to word B, even if B is not yet known at time of the analysis. There are however a few constraints for this (see annex document), and keeping left-to-right modeling, though not mandatory, will highly improve performances. As such, define variables and constraints as much as possible as if reading the sequence from left to right.

2.5 Constraint

Constraints are used on a model to specify what is expected on a word to match against the sequence.

String constraints

The string constraints are constraints applying on the sequence itself. In this group, one can find position or size constraints, as well as a content constraint to specify that we expect a specific group of nucleotids (or nucleic acids).

Struct constraints

The struct constraints applies on the word to match. It focuses on acceptable errors to match against the sequence. Both substitutions and distance errors are supported.

2.6 Morphisms

Morphisms define a transformation rule on a word. A classical morphism in biology is the complement, or the reverse complement of a DNA sequence.

The grammar introduce hard-coded transformation rules, but also a way to define its own transformation rules.

2.7 Macro controls

Macro controls add the possibility to filter the results based on a mathematical expression using a mix of variables, defined in models.

Using macro controls, one can check, for example, the global percentage of substitution on two or more variables.

Those are defined separately from the model, and are checked only when a match is found. Any variable can be used, with usual constraint operator. Only constraint is to “save” the variable.

```
controls:{
#[mod1.VAR1,mod1.VAR2]=6    // The cumulated length of VAR1 and VAR2 is equal to 6
}
mod1()==>"ccc",X1:{#[2,4],_VAR1},X2:{#[2,4],_VAR2}
mod1()==*>SEQ1
```

Texte 2: Macro controls

3 Language operators

The language supports the AND, OR, and OVERLAP operators in the pattern definition. This means that several matches are possible for a same word (besides constraints). It also support loops for the repeat operations.

The AND operator (“,” in grammar) suppose that patterns managed by AND are consecutive in the sequence. (A AND B will match in sequence for AB).

The OR operator (“|” in grammar) introduce different match possibilities for a pattern.

The OVERLAP operator (“;” in grammar) provide the possibility to search for a pattern that overlaps the previous match (example match acgt with overlap on sequence, while previous match was acac: ac**ac gt**)

The REPEAT operator (“repeat” in grammar) will search for a pattern several times up to the limit if any.

```
LTR is a good example:
sequence:  LTR ----------(spacer)---------- LTR
First matched LTR can be saved, and used later on in the sequence to find the same nucleotids suite.
```

Texte 3: Variable usage example

Variables can be used to save a matched word, so that it can be reused in the model. This is useful for word repetitions with spacers between (see word definition chapter). Those variables can be used with + and – operators in constraints. To call such variables, use the adequate call accessor (see constraints chapters, each constraint define an accessor), and optionally mathematical operators.

3.1 AND operator

To look for “aaa acgt” in sequence, “aaa” AND “acgt” is written:

mod1()==>"aaa",”acgt”

With an other model or a view

mod1()==>"aaa",mod2()

mod1()==>"aaa",(“ac”,”gt”)

Texte 4: AND operator

In LogolDesigner, simply select the Connect tool in the toolbar, click on source component and drag to target component.

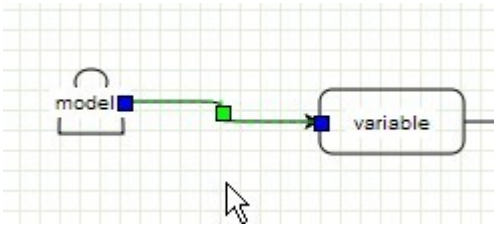

Illustration 2: LogolDesigner AND operator

3.2 OR operator

The OR is used to introduce multiple cases ni a sub-pattern, e.g. match can be A or B.

The OR operator must enclose its components in a view (e.g. parenthesis)

To look for “aaa” OR “acgt”, one should write:

mod1()==>("aaa"|”acgt”)

With an other model

mod1()==>("aaa"|mod2())

Texte 5: OR operator

In LogolDesigner, the OR operator is managed with *Fork* and *Merge* components.

At the time of the multiple choice, connect to a *Fork* element, then add the different branches to *Variables*, *Models*, *Views*.. At the end of the OR, connect all branches back to a *Merge* component.

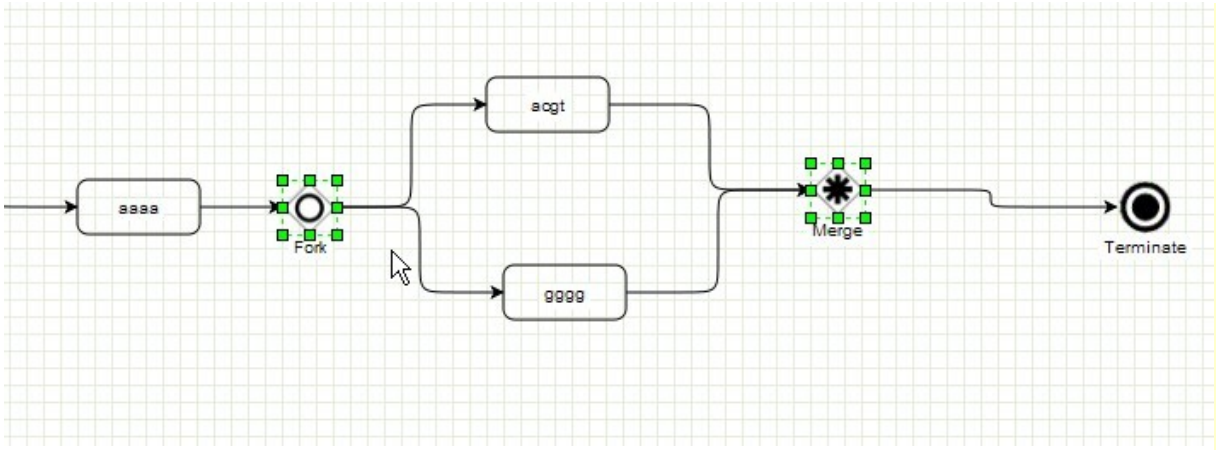

Illustration 3: LogolDesigner OR operator

3.3 OVERLAP operator

The OVERLAP operator is managed like a AND operator.

To look for “aaa” AND “acgt”, ALLOW OVERLAP, one should write:

mod1()==>"aaa";”acgt”

It will match on sequences: aaacgt (overlap) , aaaacgt (no overlap)

With an other model

mod1()==>"aaa";mod2()

Texte 6: OVERLAP operator

In LogolDesigner, to introduce overlap, just check the checkbox in the properties of the component.

3.4 Repeat operator

A repeat allows to search for a pattern several consecutive times. It allows spacer or overlap between each repeat.

Example:

I search for ccggc with 1 spacer allowed, repeated twice. It will match on ….**ccggc** a **ccggc**

I search for ccggc with overlap allowed, repeated twice. It will match on ….**ccggccggc**

I search for acgt, with [0,1] spacers, repeated up to 2 times
mod1()==>"aaa",repeat("acgt" ,[0,1])+[0,2]
mod1()==*>SEQ1
repeat("acgt")+ : no limit on repetitions, no spacer, no overlap
repeat("acgt" ,[0,2])+ : no limit on repetitions, with spacer
repeat("acgt" ;[0,2])+ : no limit on repetitions, with overlap
repeat("acgt")+[2,2] : repetited exactly twice

Texte 7: REPEAT operator

In LogolDesigner, the repeat is managed by the *Repeat* component and a *Connect* loop.

Properties of the Repeat component will define number of repetitions, overlap etc...

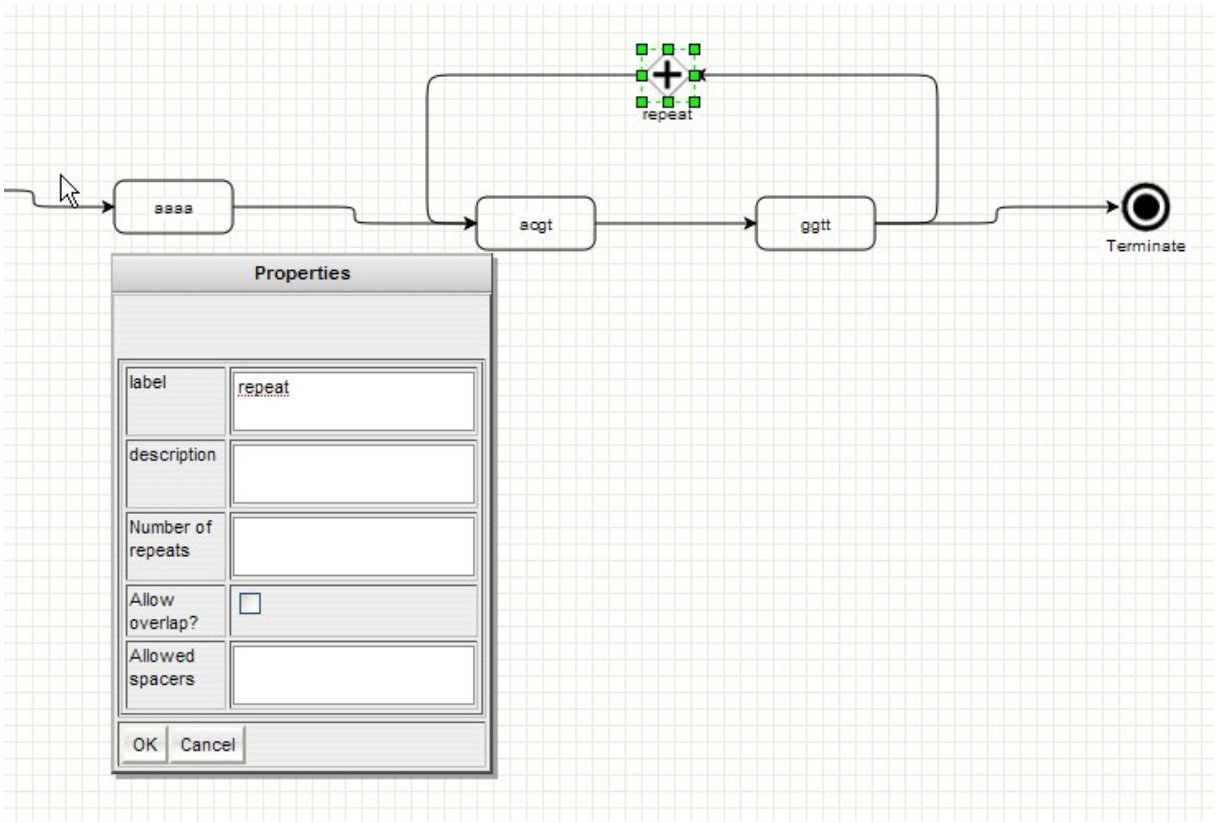

Illustration 4: LogolDesigner repeat operator

To create the loop, insert the *Repeat* component. Then connect you pattern to loop to the *Repeat* component. Once done, *Connect* the *Repeat* component back to the pattern. Connect lines can be selected to be redrawn if required.

Note: A repeat operator can apply on several components.

3.5 MATH operators

The grammar supports PLUS and MINUS operators on variables. In a constraint, to use a math operator, one should write for example:

Start position of VAR1 plus 10 = @VAR1 + 10

Math is supported only between a variable and an integer, not between two variables. The variable name is always the first element of the operation.

In Logol grammar file, one should take care to put whitespaces before and after the math operator. This is not required in LogolDesigner.

3.6 Range operator

Most constraints work with ranges as input. This is helpful to search for example a position range, or allow between a min and a max of errors.

I search for aaa with a maximum of 2 errors.

"aaa":{$[0,2]}

I search for “aaa” between the position of VAR1 plus 1000 and position of VAR1 plus 2000

“aaa”:{@[@VAR1 + 1000,@VAR1 + 2000]}

Texte 8: RANGE operator

In LogolDesigner, the range are written with a comma separator. By default, if only one expression is set, the range used is [0 , expression].

In the above example, the constraint is position of [0,@VAR1 + 2000]

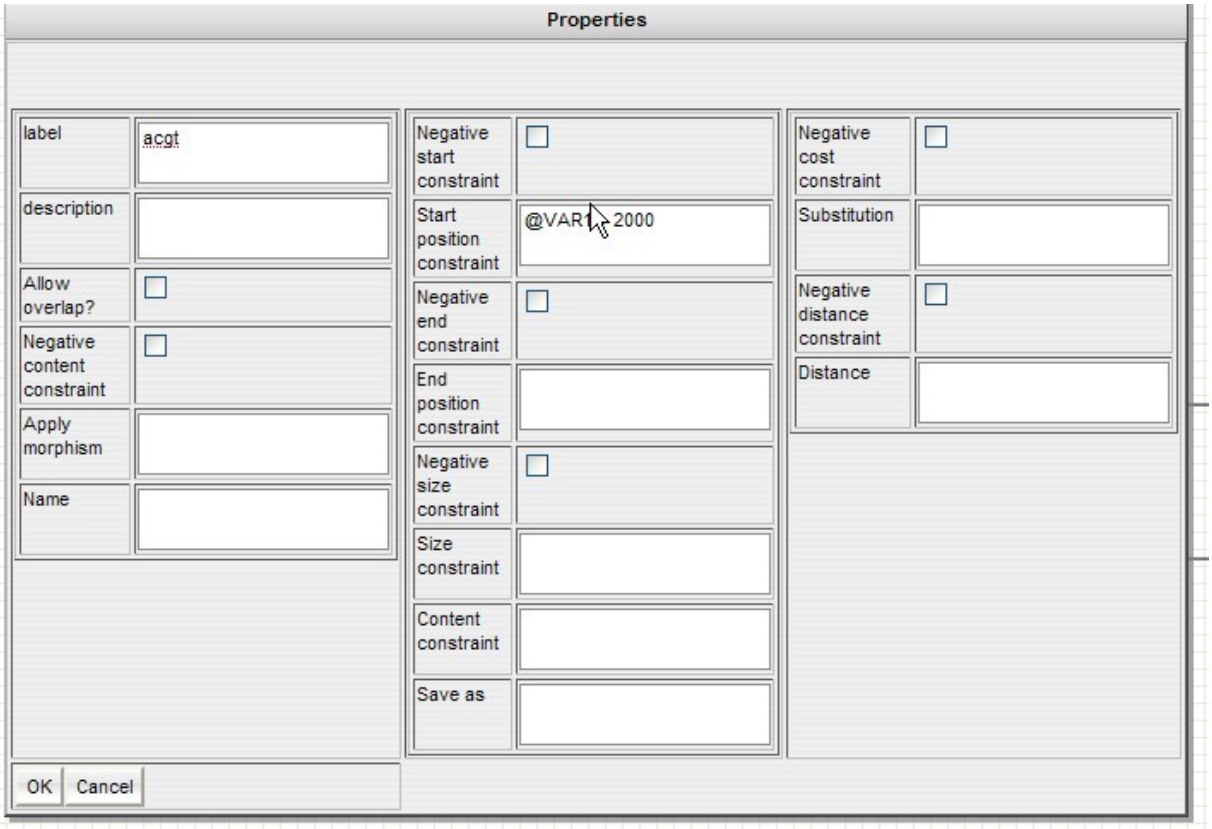

Illustration 5: LogolDesigner range operator

Additionally, if the accessor of the variable is not set in the properties, the default will be the accessor of the property. E.g., if VAR1 is called in begin property, the designer will suppose that begin property of VAR1 is used. Same for other cases.

If end position of VAR1 needs to be used in begin property, then the accessor MUST be used:

$VAR1+2000.

4 Writing rules

4.1 Language specific

If using a language file instead of graphical interface, a number of writing rules must be followed. If not compliant, LogolMatch will fail at analyzing it. The exact language rules can be found in the Logol Grammar document.

4.2 LogolDesigner writing rules

In LogolDesigner, writing rules are far less restrictive. However, for components names and data, do not use special characters nor white spaces. Accepted characters are: [a-z],[A-Z],[0-9],[-,_].

In properties of components, *Label* and *Description* fields are common to all components. The *Label* should be a simple, short, name that will appear on the component to distinguish it

in the interface. The *Description* field can hold longer comments to specify the usage or context of the component for easier understanding of the model.

Constants must be enclosed by double quotes, e.g., “acgt”

5 Pattern design

5.1 First model

Before starting a new pattern model, one needs a base template to startup.

In LogolDesigner, simply open the designer and click on “Load Example”.

This sample example gives a basic template to develop new pattern models.

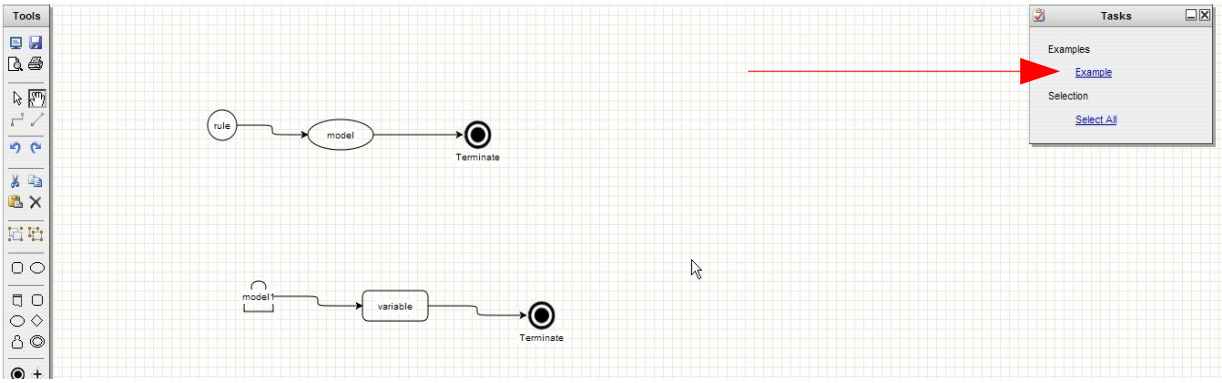

Illustration 6: LogolDesigner template load

To start with a grammar language file, here is a basic structure:

```
def:{
}
mod1()==>"aaa"
mod1()==*>SEQ1
```

Texte 9: Logol grammar template

5.2 Rule definition

As described before, the rule is the entry point.

In the template, a default rule calling one model is used. It is however possible to call multiple models (no limit) in a rule, with the possibility to transfer one or more variables from one model to the others.

Note: in a rule definition, the model order is important. The first model will be scanned against the sequence, then the second etc...

Note: Only 1 rule must be created in a model. If several rules are created, an error will be raised when executing the model.

```
mod1(VAR1).mod2().mod3(VAR1)==*>SEQ1
```

In this example, we call model mod1. This model returns a variable(word) VAR1. Then we call model mod2, then model mod3. The last model will use the variable VAR1 as input.

Texte 10: Rule definition example

In LogolDesigner,to create a rule, simply drag *Rule* component, and connect it to *Model* components. The properties of the *Rule* offer a description field to store some comments.

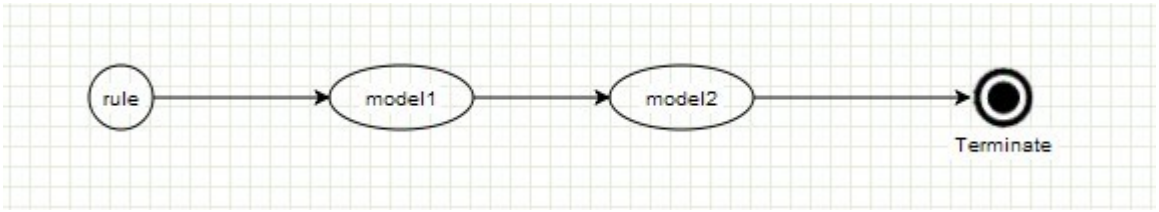

Illustration 7: LogolDesigner rule creation

Variables are specified in the model definition. To do so, edit the properties of the *Model*, and add some variable names separated by commas.

5.3 Model definition

Models are defined in *Rule*, and can be called within other *Model*.

Note: model variables type can be input or output. However it cannot be both, e.g. a variable cannot be modified once set.

Model call

A model is called by its name and, optionally, can use variables as input or output.

In LogolDesigner:

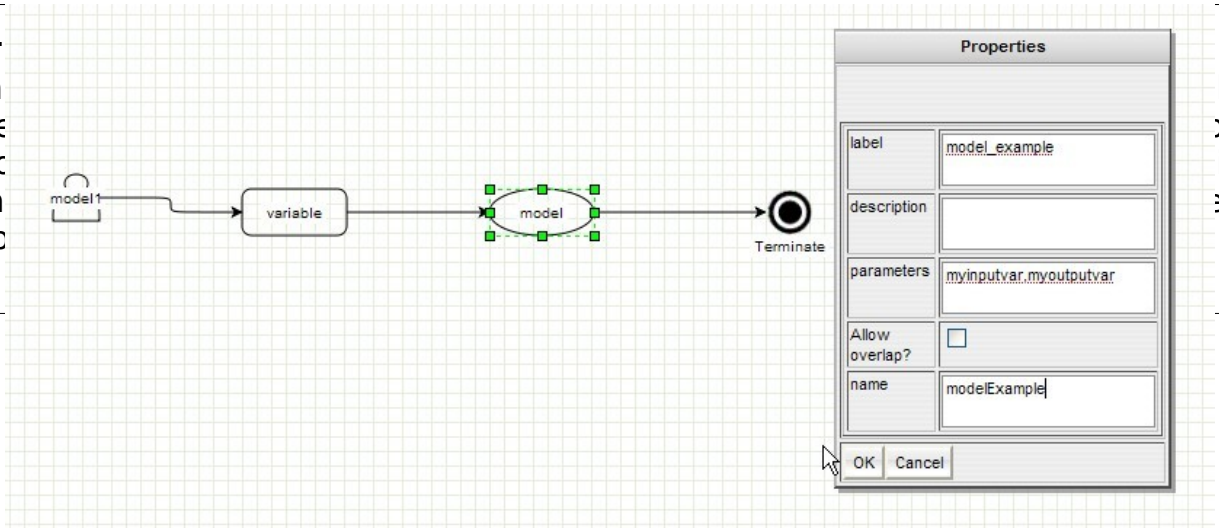

Illustration 8: LogolDesigner model call example

Model creation

To call a model, a model must be defined. There is no order for declarations. A model can first be called, then be created later on in the model. Take great care to use the exact same name between model call and model definition. Variables do not need to get the same name, they are associated by their order in the parameters field.

```
mod1(VAR1,VAR2)==>"aaa",...
```

Texte 12: model creation

A model is defined by its name and parameters. Then the pattern for this model is defined.

In LogolDesigner, a model definition must always end with a *Terminate* component.

Note: many components can end on the same *Terminate* component.

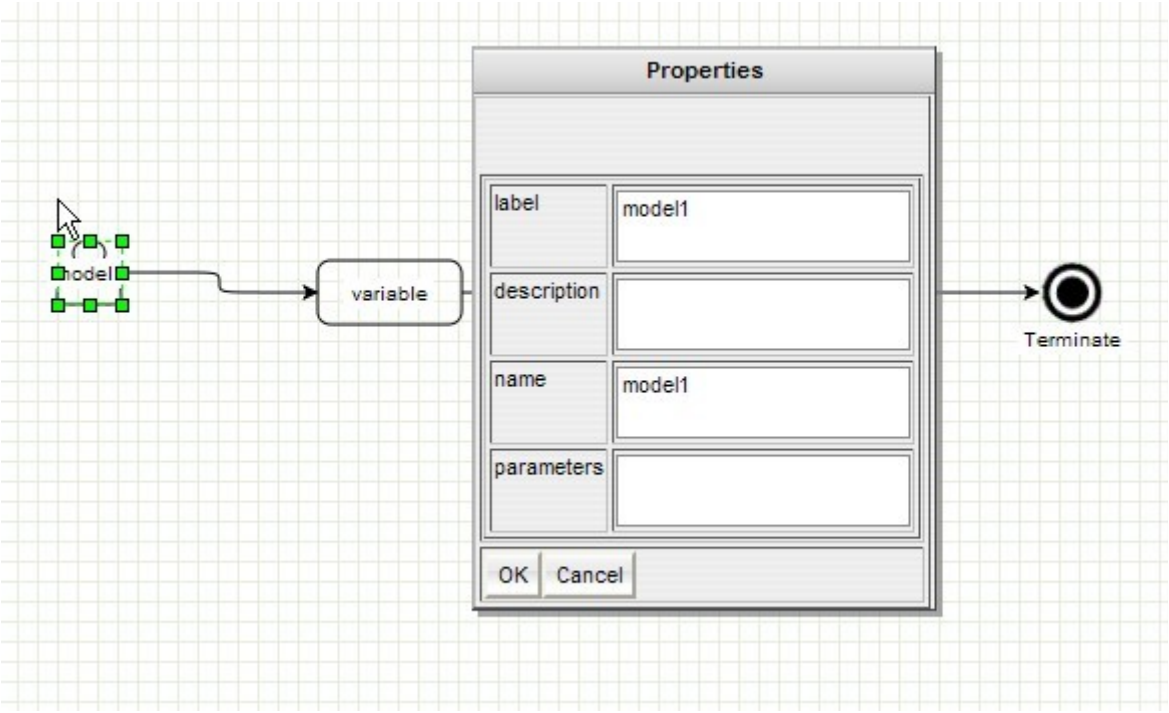

Illustration 9: LogolDesigner model definition

5.4 Entity matching

Entity (word) match is the atomic component of the grammar. An entity specifies a letter or a group of letter expected in a sequence. Constraints will apply on this word to keep “elasticity” between the word searched and the match.

Typically, an entity will be one or more nucleotids (for a dna sequence).

A word is defined by its content and its constraints (see next chapters).

If the search word is well defined, it will be defined by its content, e.g. a constant (“acgt” for example” or an other word reference (I want the same content that known word Y).

Structure of an entity:

variable_name : { string constraints } : { struct constraints }

Grammar usage:

mod1(VAR1)==>"aaa", ?VAR1, VAR2:{string constraints}:{struct constraints}

A word can be set explicitly, e.g. “aaa”, by a variable name (?VAR1 means I want to get content of VAR1) optionally followed by additional constraints, or defined by a new name followed by constraints (VAR2).

Here follows several examples:

“acgt”

?VAR1:{ $[0,2]}

VAR2:{@[1000,1500],_SAVEDVAR2}

VAR2:{?VAR1}:{$[0,2]}

Texte 13: word definition

LogolDesigner word usage:

It is defined by the *Variable* component. In the case of defined content, you don't need to fill the *name* property, it is optional. The content will be managed by the content constraint.

If word is not defined by its content (only by its size for example, or for parental comparisons), then fill the *name* property by a variable name.

Note: if content is a constant e.g. “acgt” or a *Definition*, then it can be set directly in name, like in below example. This is an additional writing possibility but content can always be set in Content constraint (see later). If case of a string constant, it should be enclosed by double quotes (“acgt”). For a definition, simply put the name of the definition.

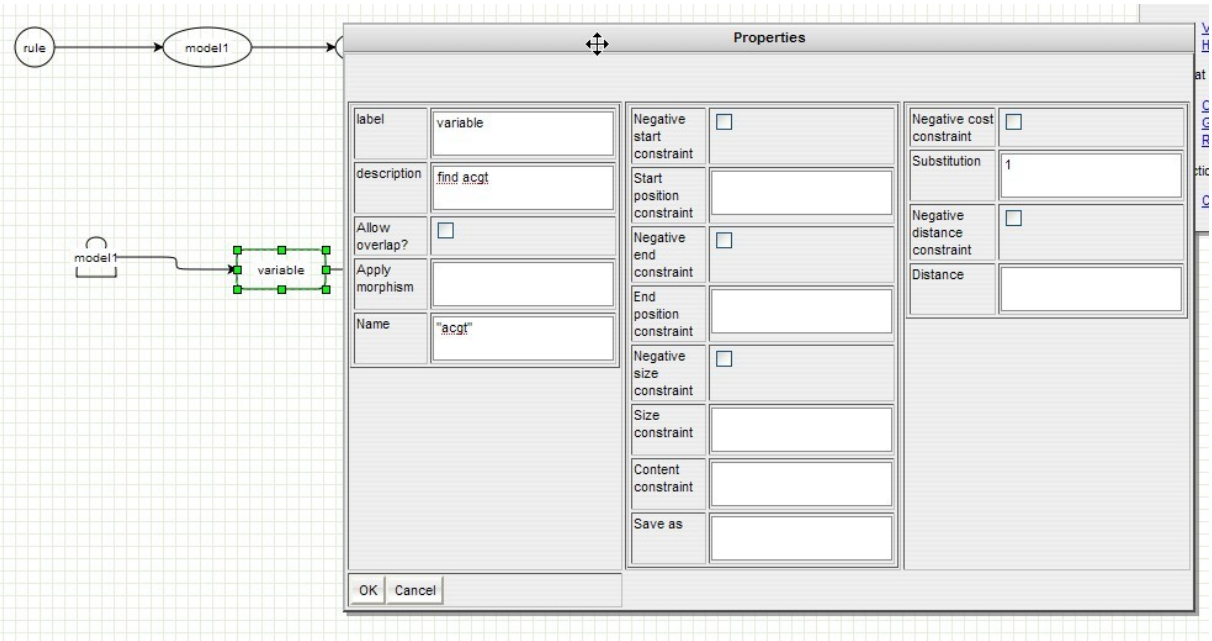

Illustration 10: Logol Designer word definition

A few examples:

Use case	Name property
I want to find “acgt”	“acgt”
I want to find a START (defined in a definition)	START
I want to find X content	None (will fill constraints)
I want to define new variable X, no parental comparisons expected	None (will fill constraints if variable needs to be saved for later use)
I want to define new variable X, future parental comparisons expected	X

5.5 Views

The views are a way to group some models, words etc..

The goal can be double. At first, it provides a logical grouping of some components, and a way to get the global match of this group in the results. Secondly, it is a way to apply some constraints (see constraints chapters) on a group of components.

First case:

If A and B are in the same group, result match will give the (A,B) result (starts at A and ends at B, with A+B errors), with of course the details of A and B.

Second case:

I look for A with 2 errors max, then I look for B with 2 errors max. I want that A+B errors is less than or equal to 2. (A,B):{$[0,2]}

```
mod1(VAR1)==>("aaa", "ccc"):{string constraints}:{struct constraints}
A view is characterized by parenthesis. Then constraints can be applied to the group
```

Texte 14: view definition

In LogolDesigner, the first case (logical group) is also a way to get an analytic view of the model help with the collapse/extend functions of the *View* component.

Simply drag a *View* component in the main window, then drag some components into the *View* component (a *Model*, or a *Variable*).

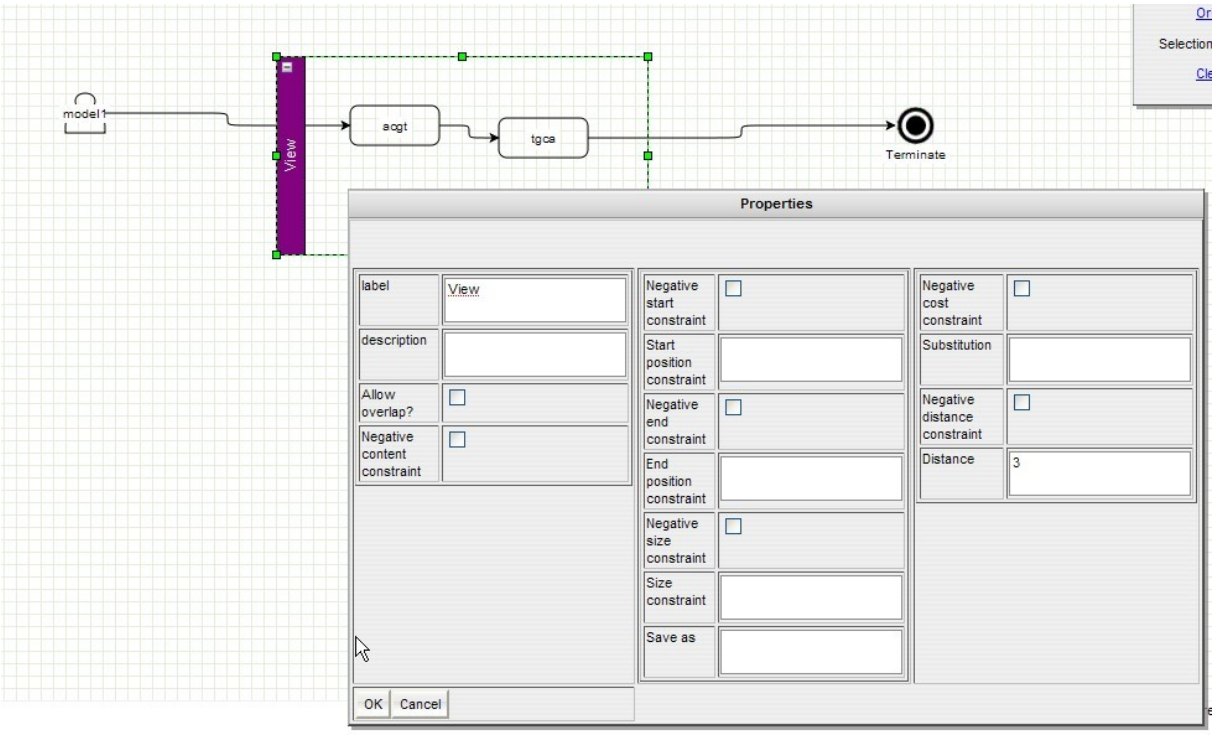

Illustration 11: LogolDesigner view definition

In the above example, we set a distance constraint on the group.

The collapse button can reduce the block to hide the internal components of the view.

A right click on the view show the “Go into” menu. If selected, the browser opens the content of the view. A right click again, will then show the “Go up” menu to go back to the main model. Multiple layers are managed for easier management, mainly in the case of complex views where number of components is important and would be difficult to manage in the main model.

Do not forget to link to the first component of the view and to link to the last component of the view (views are connected by its internal components, not by the view component itself).

5.6 Spacers (gaps)

Spacers are a way to define a gap between two entities. Spacer can be of undefined length or constrained by a size range, as well as by its position in the sequence. Spacer cannot be saved, it is only to specify that next entity is further in the sequence. It is preferable to place spacers in front of known entities rather than a *Model* for example, or a *Fork.*

```
mod1(VAR1)==>"aaa", .*:{#[200,500]},"ccc"
```

“aaa” and “ccc” are separated by a gap of size >200 and <500.

Gap is specified with the characters “.*”

Texte 15: spacer definition

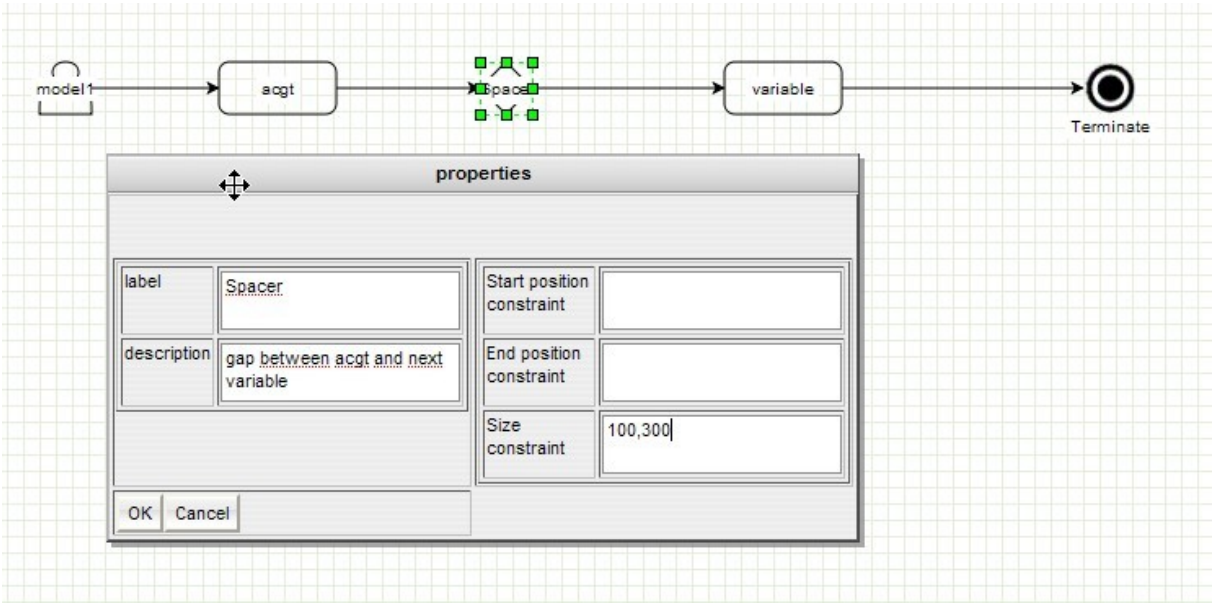

Illustration 12: LogolDesigner spacer usage

5.7 String constraints

Constraints applies on the match of the pattern. The result must match the applied constraints.

Some of those can be negative constraints, e.g. , the constraint must fail to match.

Caution: an absolute position will not work if sequence is analyzed in both direction (execution option). In such a case , use only relative position constraints.

Constraints are not exclusive, but only one of each can be defined (1 begin and 1 content are allowed, while 2 begin are forbidden).

Range in stringconstraints can define a “no limit” element with the character “_” (file or designer). For example #[10,_] would mean a size constraint of at least 10.

Begin constraint

It will check the start position in a left to right reading of the sequence. It is a range.

The accessor is character “@”.

Begin position of variable VAR1 is @VAR1.

End constraint

It will check the end position in a left to right reading of the sequence. It is a range.

The accessor is character “@@”.

End position of variable VAR1 is @@VAR1.

Size constraint

It will check the size of the match. It is a range

The accessor is character “#”.

The size of variable VAR1 is #VAR1.

The size constraint can be set as optimal, e.g. try to search for the largest match for this variable with the accessor: “#OPT”.

Optimal search filter can be set on several variables. The optimal solution is the longest match for the variable. For several optimal variables, the program search for the longest match of a variable where it is minimal for the other ones.

Content constraint

The content constraint try to match a specific content/word on the sequence. It can be used alone to search for exact match, or combined with struct constraints to introduce errors.

The accessor is character “?”.

The content of variable VAR1 is ?VAR1.

Save constraint

Saves the content of the match on the sequence, so that it can be resused in an other component with the content constraint. There is no accessor for this constraint. It is accessed with the Content constraint accessor.

```
mod1()==>("aaa"):{_VAR1}:{$[0,1]}
I look for aaa, with 1 error allowed, and save result in VAR1
```

Texte 16: save constraint

5.8 Structure constraints

Constraints applies on the structure of the pattern. The result must match the applied constraints. If the variable is saved, the cost and distance found between the sequence and the pattern will be available via their accessor.

Some of those can be negative constraints, e.g. , the constraint must fail to match.

Cost constraint

This constraint refers to allowed substitutions in the sequence compared to the pattern. It is a range.

The accessor is character “$”.

The cost found for variable VAR1 is $VAR1.

Distance constraint

This constraint refers to allowed edit distance in the sequence compared to the pattern. It is a range.

The accessor is character “$$”.

The distance found for variable VAR1 is $$VAR1.

Alphabet constraint

This constraint is a filter constraint. It means that it can be used to filter a match, but the value of the filter will not be stored in results.

It specifies the minimum percentage required between a match and a defined alphabet.

It is defined by: % ”xxxx”:INT

xxx are the expected characters to match and INT is the minimum percentage.

Warning: A whitespace is required between % and double quote.

Example:

% ”cg”:70 expects to match at least 70% of G and C nucleotides.

5.9 Morphisms (and modifiers)

Morphisms are transformation rules for a pattern. When a pattern is known (“acgt” or an other variable content), it can be transformed help with a modifier. The *modifier* calls the *Morphism*

element to apply its rules, and then the tool tries to match the result of the transformation with the sequence with the constraints of the entity.

Modifiers applies on *Entities* only. They should be prepended by a “-” to specify that element should be reversed before applying the modifier or a “+” for direct apply.

To define a morphism and use it as a modifier:

Morphism rules do not limit to one character, several characters can be mapped to a single one (example: morphism(foo,at,g), will map at to g)

```
def:{
morphism(foo,a,g)
}
mod1()==>"aaa",-"foo" "tgca"


mod1()==*>SEQ1
```

In this example, we define a morphism called foo, which maps “a” to “g”.

The tool will take “tgca”, reverse it (“acgt”), and apply the morphism “gcgt”. The searched word is “gcgt”.

Texte 17: morhpisms usage

<u>Warning</u>: All transformation rules must be defined in the morphism, even if there is no mapping, e.g. morphism(foo,a,a)

<u>Already defined morphisms:</u>

- *wc* : word complement, available with + and – directions.
- *p2d*: protein to dna conversion. In a nucleic search, it allows to specify a proteic string that will be converted in its nucleic possible matches. Example: the “f” will be mapped to “ttt” or “ttc”.
- *reverse*: simply reverse the string. Direction is not taken into account, though for better lisibility, it is advised to prepend it by a minus sign.

In LogolDesigner, add as many *Morphism* components as required to create the whole rule. Specify the morphism name that will be used for modifiers, set in “in” property the characters to transform (no quotes, no comma), and set in “out” property the expected character.

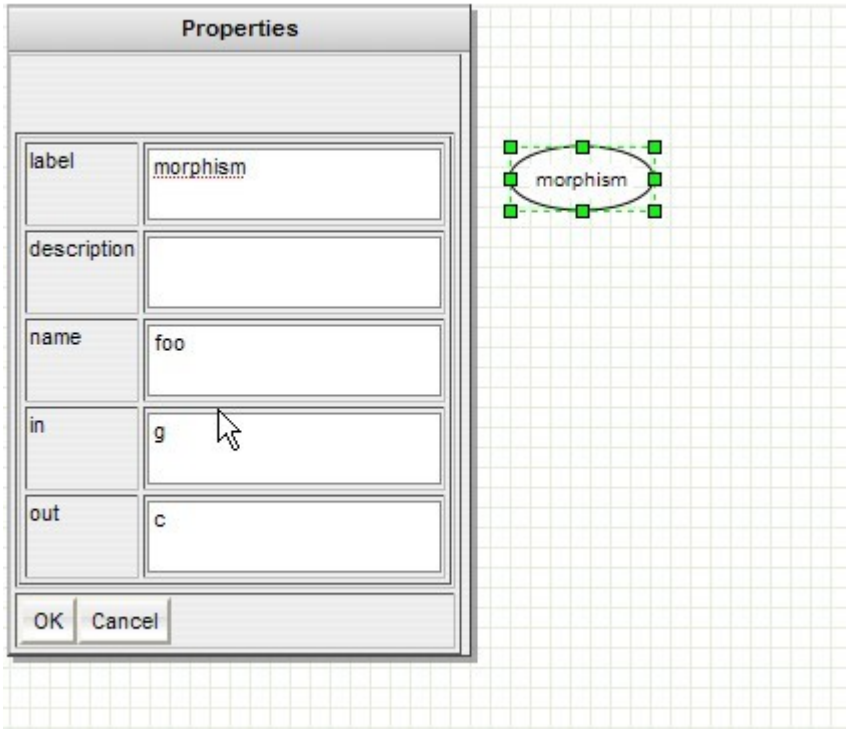

Illustration 13: LogolDesigner morphism definition

Then morphism can used used via a modifier to be applied on an *Variable*.

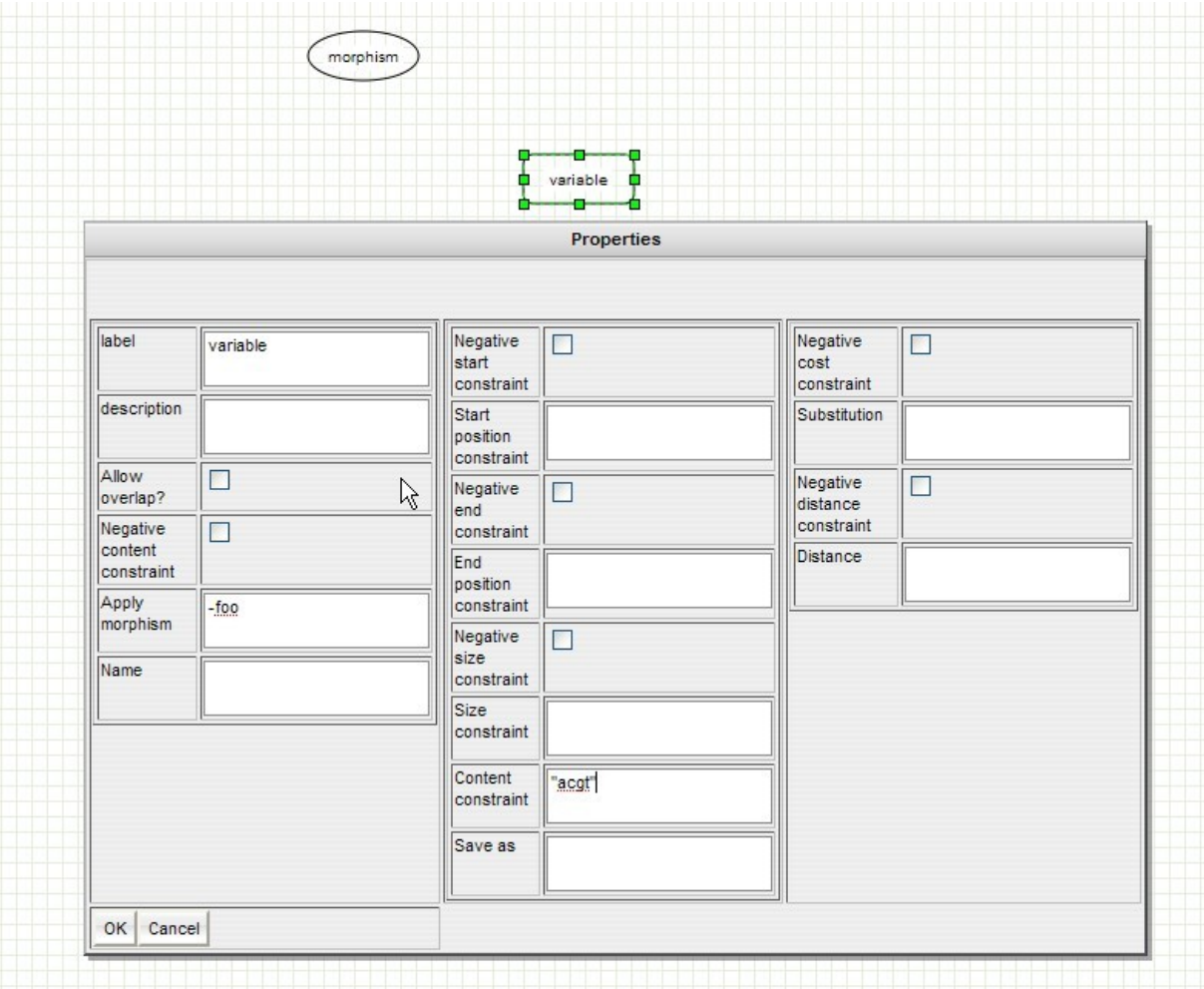

Illustration 14: LogolDesigner modifier usage

5.10 Definitions

Specific to the LogolDesigner, definitions are a way to define a string, associated with a name, to be used several times in the model. This prevents errors in copying same data in several components, and provides easier reading help with the definition name.

To add a definition, drag a Terminal to the main window, and edit properties. Label will be the name to be used as reference in other components properties, while Name will contain the data itself (“acgt” for example”).

Though Terminals do not need to be connected to any component, it is advised to connect Terminal components each other for easier reading.

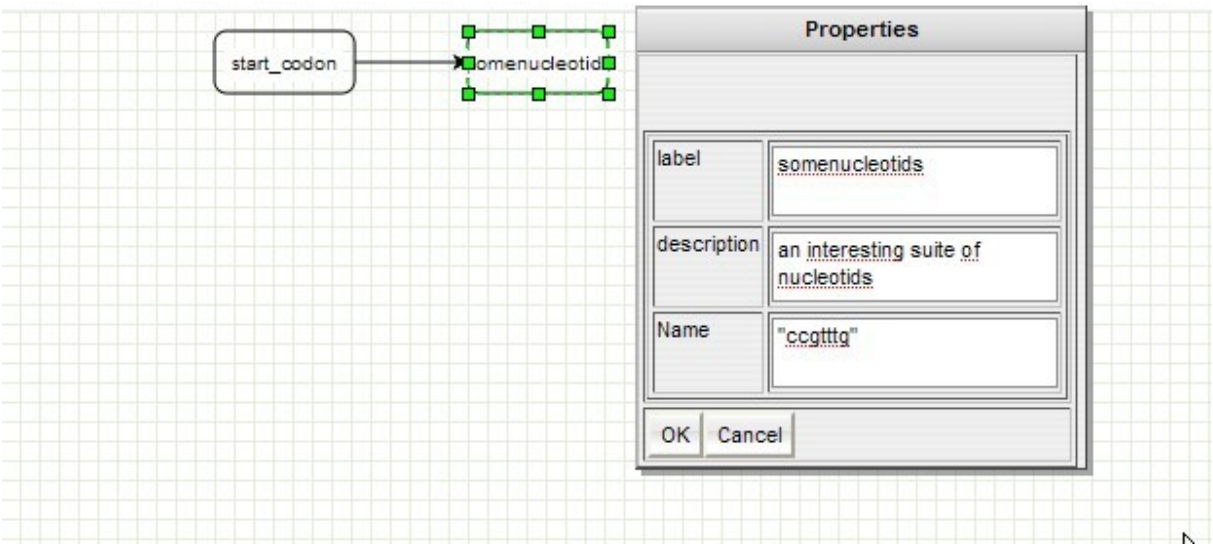

Illustration 15: LogolDesigner definition usage

In the above example, we call the modifier foo (foo morphism), to be applied on content constraint “acgt” after a reverse (minus in front of modifier).

6 LogolDesigner toolbar

Here is the description of the toolbar:

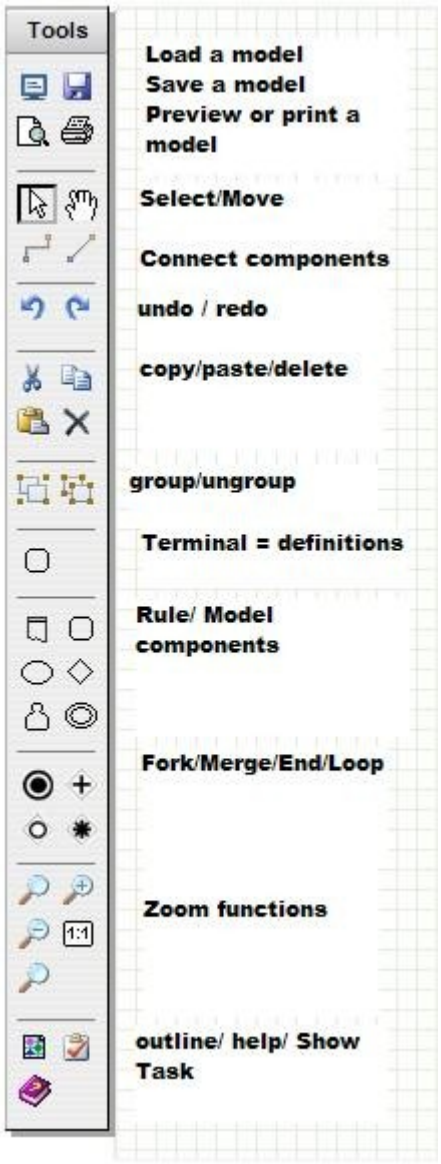

Illustration 16: LogolDesigner toolbar

7 Development tips

7.1 Start anchors

All patterns are search within the input sequence with an input gap of undetermined size:

- If match is expected to be found in a known range at the start of the sequence, specify a begin position constraint on the first block.
- Always try to set the first block with a constant value ("acgt" content constraint for example). This anchor will drastically reduce the number of possibilities, even if number of solutions is high.

7.2 Gaps / spacers

Avoid using a gap in front of a fork (OR). Prefer the fork, followed on each branch by a gap.

7.3 Views

Limit the number of views

- Using views is very useful from a graphical point of view to group the data. However grouping feature adds a logical behavior to the grammar (with possibly specific constraints, per view detail in the results) . Grouping should be limited to its role, e.g. to give a meaning to a group of data (to use it as a constraint in an other element, or get a high level view in the results)

7.4 Spacer against entities

Entity can also be used to search for a spacer with a relative small size. Spacers (see Spacer chapter), allow to search for a pattern on the sequence with a gap compared to current position. When one knows that remote pattern isquite near in the sequence, one can gain some performances by replacing a spacer with an entity. To do so, simply create an entity with a name (VAR1 for example), and set a size constraint range [0, maxsize].

7.5 Parental comparison search

Parental comparison can be useful to compare some entities having the same parents from phylogeny point of view.

It can be used to find childs that differs when being copied according to different rules (for example LTR that would get stronger conservation at one side than the other).

It could be expressed as below:

```
mod1()==>X1:{$[0,2]},...,X1:{$[0,6]},....,X1:{$[0,2]}
```

This example looks for 3 copies of X1, but the copy at the middle will have more errors than the other copies.

Texte 18: parental comparison

The parental comparison does not take one entity as reference to find the other, the reference is the old parent.

This should be used for specific cases, not to compare 2 simple entities. Parental comparison is very costly during the earch.

7.6 Negative constraints

Negative constraint give the possibility to check that word do not match a constraint. If negative constraints can be useful, one should take care at using them regarding performances. Negative constraint does not help in the search, this is only a posterior condition checked after each possible match. This means that, when looking for an entity, the negative constraints are not used. Once a match is a candidate, then and only then the negative constraint is checked.

Thus, using a negative End constraint on a well defined word could be a good candidate usage. Negative content constraints have poor performances and should be avoided if not really required.

Negative contraint is select in graphical interface via “negative ...” checkbox.

In grammar, it is set with a “!” characte, example:

```
mod1()==>”a”:{_X1},”cgt”,!?X1:{#[1,4]}
```

Meaning that after “cgt” we want anything of length [1,4] not matching X1.

If no length contraint is set, the program will suppose that we want to match the samelength than X1 but not X1.

8 Annex A: Restrictions