transsys User Manual

Jan T. Kim

kim@inb.uni-luebeck.de

Institute for Neuro- und Bioinformatics
Seelandstr. 1a
D-23569 Lübeck, Germany

December 9, 2013

1 Introduction

1.1 Motivation

During recent years, gene regulation has become a major focus in the molecular biosciences. Many important biological processes, such as cell differentiation, morphogenesis, diseases and responses to drug administration, have been found to be determined by networks of regulatory genes at the molecular level. However, there is a lack of generic and theoretical understanding of regulatory networks. Progress in this area would benefit from a concise and generic representation of regulatory networks, rendering them accessible to theoretical, statistical and modelling approaches. transsys, a contraction of "transcription factor system", is intended to be a contribution towards this goal.

1.2 Concept

Regulatory gene networks are constituted by genes encoding transcription factors and the encoded transcription factors which regulate the expression (and hence the activity) of the regulatory genes as well as the expression of other genes (which are sometimes referred to as "phenotypic realizator genes"). transsys attempts

to capture these two essential components and their interrelations. At the core, transsys is a formal language for specifying the core entities constituting regulatory networks as described above. The transsys sofware package provides a set of tools for visualizing and analyzing networks specified in the transsys language.

While a concise representation of a regulatory network may already be of some value for managing this type of biological information, a network alone is of limited use as a model of a biological system. Regulatory networks receive input from the biological system in which they are embeded. These input signals can, depending on the level of resolution or abstraction of the particular model, be thought of as other regulatory networks, terminals of intracellular signal cascades, intercellular signals, or environmental signals. Moreover, there are feedback loops between the regulatory network and the pattern formation and morphogenesis processes which unfolds in response to activities within the network.

The major goal of transsys is to capture the integration of regulatory networks within such biological contexts, as these contexts can be assumed to qualitatively determine the dynamics and phenomena emerging within a network, and moreover, the biological sense made by regulatory networks can only be captured by integratively (holistically?) modelling a network along with its biological context. To achieve this goal, transsys is designed as a component which conceptually can be embedded within a more extensive modelling context. As a first case study, transsys has been combined with the Lindenmayer system formalism. The resulting model, labelled L-transsys, is described in the last part of this report.

As the modelling level of transsys is the level of regulatory networks, surrounding levels of biological organization are not represented in much detail, e.g. because transsys does not address the metabolic level, no attempt is made to enforce energy or mass conservation. The rationale for this is the observation that transcription factor synthesis and decay accounts for a minute fraction of the entire cellular metabolism.

2 The transsys Model

Modelling with transsys involves a two-stage process: Firstly, a transsys model has to be defined. In a transsys definition, the core elements of the regulatory network are described by a set of numerical properties. For factors, these properties are the decay rate and the diffusibility. The properties of genes

are divided in a promoter section, specifying which factors have activating or repressing effects on the expression of the gene, and a product section, specifying which factor is encoded by the gene.

Once a `transsys` is specified, instances can be created. The information stored in a `transsys` instance is an array holding the concentrations of the factors of the network. The factor and gene properties specify a procedure for updating these concentrations, i.e. for computing the factor concentrations in the next time step given the concentrations in the current time step. In OO parlance, the factor concentrations may be thought of as member variables of a `transsys` instance, while the factor and gene properties specify and parameterize the method for simulating the internal processes taking place within the instance during one time step.

2.1 Factor Model

The properties of a factor in `transsys` are a decay rate and a diffusion rate. The decay rate is a value between 0 and 1 that specifies how much of the total amount of a factor decays during a time step. The diffusion rate specifies how large a fraction of a factor is distributed among the neighbouring `transsys` instances. Evidently, the existence of at least two instances that are coupled in some way is necessary in order to get any reasonable effect from diffusion. The details of this coupling cannot be deduced from the `transsys` alone, the system relies on the context model to provide multiple instances and a connection graph of some kind. As a simple example, these can be provided by a CA.

2.2 Gene Model

The gene model used in `transsys` comprises two parts, a regulatory block and a product block. The regulatory block describes how the gene's activity is affected by the various factors. The product block simply determines which factor is synthesized as the gene's product.

3 Program Overview

The `transsys` package currently comprises these programs:

- `transcheck`: Parse `transsys` code and write parsed code into output. This is used during parser developments, to ensure that all information from

the code is read in as intended. However, this program may also be of some use for locating errors in `transsys` code.

- `transexpr` Generate a time series factor concentrations. Output file format is suitable for the `gnuplot` program.
- `transps` Generate PostScript graphics of regulatory networks specified as `transsys` programs
- `transscatter` Produce scatter plots by starting off a `transsys` instance with random initial factor concentrations and performing a given number of updates. Intended to check networks for robustness and to search and characterize attractors.
- `ltrcheck` Perform some derivations of an `L-transsys` program and write each `L-transsys` string to output. Used for exploring, checking and debugging `L-transsys` programs.
- `ltransps` Produce graphical 2D rendition of `L-transsys` run in PostScript
- `ltransgl` Produce 3D rendition of `L-transsys` run using OpenGL

4 Examples

4.1 Writing a `transsys` Program

4.1.1 The Beginning

A `transsys` program consists of the keyword **`transsys`**, followed by a name and a body which is enclosed by curly braces and contains the actual components forming a regulatory network. As a start, and to set up a framework, we'll start with an empty system, i.e. with one that does not contain any components:

```
transsys example
{
}
```

4.1.2 Adding a Factor

Transcription factors are specified by **`factor`** statements in the body. So, let's start filling our emtpy system by adding a factor:

```
transsys example
{
  factor F1
  {
  }
}
```

As you see, a factor specification has the same structure as a `transsys` specification: A keyword (**`factor`**), a name (`F1`) and a block enclosed by curly braces, which is empty here but will be filled soon.

4.1.3 Adding a Gene

Our transcription factor `F1` won't be much fun unless we also provide a gene encoding it. Perhaps not surprisingly, genes are also specified by a keyword, namely **`gene`**, followed by a name and a block:

```
transsys example
{
  factor F1
  {
  }

  gene g1
  {
    promoter
    {
      constitutive: 1.0;
    }
    product
    {
      default: F1;
    }
  }
}
```

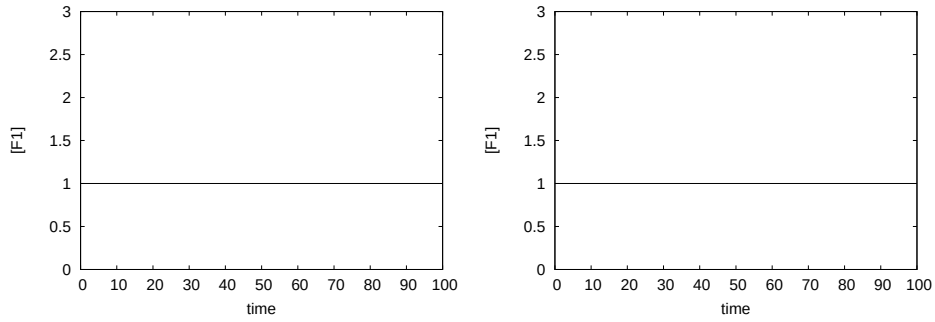

Figure 1: Temporal dynamics of factor F1 with a decay rate of 1 (left) and a decay rate of 0.5 (right).

4.1.4 Trying it out

Our example transsys program is not (yet) anything interesting as a biological model or a complex system. But we can use some transsys programs on it for demonstration purposes. You can create a time course of the concentration of F1 with the command

```
% transexpr example.tra example.plt
```

The resulting file can be plotted using gnuplot:

```
% gnuplot
gnuplot> plot 'example.plt' using 1:2 with lines
```

This plot, shown in Fig. 1, will show you that the concentration of F1 is 1 at all time steps excepting the initial one. This may not seem quite right to you: If 1.0 is the amount of factor synthesized from g1 in each time step, should it not accumulate over time? The answer is that it could, if we had specified a decay rate. Let's use this occasion to introduce the transcheck program for revealing the default factor properties:

```
% transcheck example.tra
```

gives you the an output of our example transsys program in which the F1 block reads

```
factor F1
{
```

```
    decay: 1;
    diffusibility: 1;
  }
```

As we see, each factor has two properties, a decay rate and a diffusion rate. If no specification is given in a factor's block, `transsys` assumes the parameter to be 1. A decay rate of 1 means that all factor molecules (i.e. 100% of them) decay within one time step. This explains why the factor concentration does not exceed 1.

4.1.5 Specifying a Decay Rate

It would not make any sense if the decay rate could not be changed, and quite obviously, this is done as seen in the `transcheck` output shown above. So, let's specify the decay rate explicitly:

```
factor F1
{
  decay: 0.5;
}
```

If you run `transexpr` on the `transsys` program with the factor definition modified this way, you'll see that `F1` indeed accumulates over time, approaching a limit of 2 asymptotically, see Fig. 1.

4.1.6 Controlling a Gene's Expression

In the example `transsys` program developed so far, the gene `g1` is constitutively expressed. Now, it's time to take a central step towards modelling regulatory networks by introducing another gene controlled by `F1`. We do this by adding the following to our `transsys` program:

```
factor F2 { decay: 1; }

gene g2
{
  promoter
  {
    F1: activate(2, 5);
  }
```

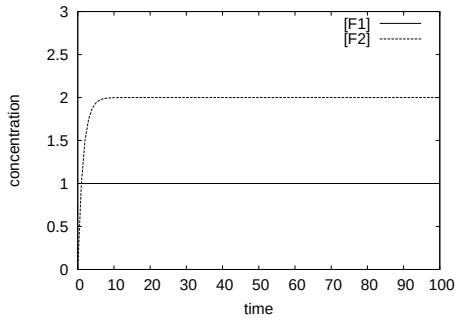

Figure 2: Dynamics of concentration of F1 and F2 in a transsys program where F1 regulates expression of the gene encoding F2.

```
    product
    {
      default: F2;
    }
}
```

You can check the resulting dynamics in factor concentrations by running:

```
% transexpr example.tra example.plt
% gnuplot
gnuplot> plot 'example.plt' using 1:2 with lines
gnuplot> replot 'example.plt' using 1:3 with lines
```

The results are also shown in Fig. 2. The dynamics of F1 are as in the preceding versions of the example. To understand the dynamics of F2, notice first that the decay rate was set explicitly to 1. Thus, the total amount of F2 in a time step is equal to the amount synthesized in that time step.

Regulation of g2 by F1 is modelled by the activate statement. The factor activating the gene is mentioned before the keyword activate. Quantitatively, activation is computed according to the Michaelis-Menten-equation. The two parameters of activate, which are denoted by $a_{\mathrm{spec}}$and $a_{\max}$, are analogous to the Michaelis-Menten parameter K_M and $v_{\max}$, respectively. Thus, as the concentration of F1 approaches 2, the rate of synthesis of F2 approaches $\frac{a_{\max} \cdot c_{F1}}{a_{\mathrm{spec}} + c_{F1}} = \frac{5 \cdot 2}{2+2} = 2.5$.

Fig. 2 shows that synthesis of F2 starts one time step after F1 begins to accumulate. This is due to the implementation of transsys: Factor synthesized in a

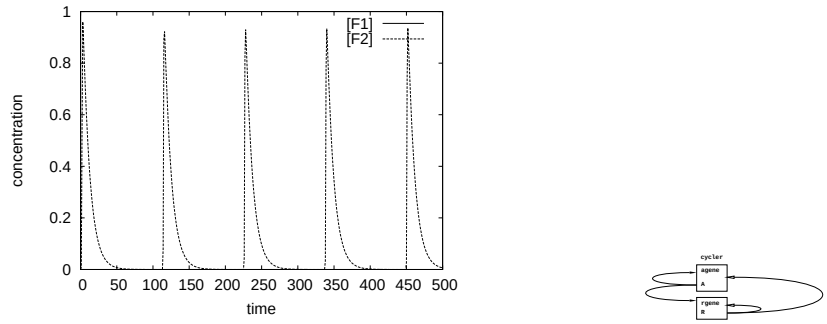

Figure 3: Oscillatory dynamics in a regulatory network of two genes (left) and a graph representation of the regulatory network (right).

time step does not participate in gene regulation in that time step, effects set in in the subsequent time step.

4.1.7 Getting Advanced: Autoregulation

Having seen how one gene can be regulated by a factor encoded by another gene, the stage is now ready for presenting a `transsys` program that deserves being called a regulatory network:

```
transsys cycler
{
  factor A { decay: 0.1; }
  factor R { decay: 0.1; }

  gene agene
  {
    promoter
    {
      constitutive: 0.01;
      A: activate(0.01, 1.0);
      R: repress(0.1, 1.0);
    }
    product
    {
      default: A;
    }
```

```
  }

  gene rgene
  {
    promoter
    {
      A: activate(1.0, 10.0);
      R: repress(1.0, 1.0);
    }
    product
    {
      default: R;
    }
  }
}
```

Fig. 3 shows the temporal dynamics of the two factors `A`, which activates both genes and `R`, which represses both genes. Initially, `agene` is slightly activated due to the `constitutive` statement in the promoter. Accumulation of `A` amplifies activation of `agene`, but also results in activation of `rgene`, which encodes `R`. The parameters of activation and repression are chosen such that the repression effects of `R` eventually lead to a temporary shutdown of expression of both genes. After that, decay of `R` finally allows constitutive expression of `agene` to set in again, which starts the next oscillation.

4.1.8 Visualizing the Regulatory Network

The `transsys` program introduced above is surely simple enough to be understandable by just reading the code. Larger networks, however, demand larger and more complex `transsys` programs. The program `transps` can be used to graphically render the regulatory network structures encoded in a `transsys` program:

```
% transps example.tra example.ps
```

For the simple autoregulatory network shown above, the corresponding graph is shown in Fig. 3.

4.2 Writing a `L-transsys` Program

4.2.1 The Beginning

`L-transsys` uses the same principles as `transsys`, in particular, the concept of named blocks was applied in `L-transsys` whereever it seemed reasonable. Thus, an empty `L-transsys` looks like this:

```
lsys example
{
}
```

4.2.2 Defining a Symbols

Symbols (from which strings are assembled) are the basic unit of L-systems. In `L-transsys` symbols have to be defined before they can be used. So, let's define a symbol:

```
lsys example
{
  symbol shoot_piece;
}
```

4.2.3 Defining a Rule

Rules are the centerpiece of traditional L-systems, and in `L-transsys`, they are mechanism which links realization of morphogenetic processes to gene expression. But we'll postpone introduction of `transsys` programs with `L-transsys` a little in order to first demonstrate the basics of rule definition. A simple example rule reads:

```
lsys example
{
  symbol shoot_piece;

  rule doubleshoot
  {
    shoot_piece --> shoot_piece shoot_piece
  }
}
```

As you might guess (at least, if you have some experience with L-systems) this rule replaces each occurrence of a `shoot_piece` symbol with a string of two `shoot_piece` symbols. Repeated application yields a string of `shoot_piece` symbols which exponentially grows in length.

4.2.4 Trying it out

To actually see the rapidly growing string, it is necessary to provide the system with an initial string to start with. This initial string is called the axiom in L-system terminology, therefore, `L-transsys` uses a keyword called `axiom` for this purpose:

```
lsys example
{
  symbol shoot_piece;

  axiom shoot_piece;

  rule doubleshoot
  {
    shoot_piece --> shoot_piece shoot_piece
  }
}
```

This `L-transsys` program is certainly minimalistic, and it's not original at all as it doesn't use anything `transsys` specific, but let's use it to see `L-transsys` at work nonetheless:

```
% ltrcheck -n 5 example.trl
```

This command prints out the result of the first 5 derivations of the axiom. The output is not really interesting, consisting just of lots of `shoot_piece`s, so it's not shown here.

4.2.5 Adding Graphics

The output not shown above is not just boring because the `L-transsys` program comprises just one symbol. String dumps are useful for debugging, but the real strength of L-systems can only be seen if the string is rendered graphically. Now, since `L-transsys` lets us define arbitrary symbols, it cannot know how what

graphics we want to associate with our symbols. Thus, we end up having to specify that too:

```
graphics
{
  shoot_piece
  {
    move(0.5);
    color(0.7, 0.7, 0.7);
    cylinder(0.1, 1.0);
    move(0.5);
  }
}
```

By adding this piece to the `L-transsys` program developed so far, we obtain a minimalistic system which can be used for 3D rendering with the program `ltransgl`:

```
% ltransgl example.trl
```

Running this program brings up an OpenGL window in which a grey cylinder is displayed. The object can be moved along the X axis by dragging with the left mouse button. Translation along the Y and Z axes are possible by pressing and holding down the shift or control key, respectively. The object can also be rotated around all axes by dragging with the right mouse button, again using the shift or control key to select the Y or Z axis. Pressing `n` computes shows the next derivation step, `p` shows the preceding step. The `Home` and `End` keys move to the initial step (i.e. the axiom) and the last step computed so far, respectively.

4.2.6 Changing Direction

Shoot pieces forming a straight chain seem somewhat unflexible. Of course, `L-transsys` allows changes in orientation. There are three axes around which one can rotate in 3D, and rotations are done with the graphics functions `turn()`, `roll()` and `bank()`. Let's demonstrate the `turn()` function with an example:

```
lsys example
{
  symbol shoot_piece;
  symbol left;
```

Figure 4: Demo of `turn()` graphics command.

```
  symbol right;

  axiom shoot_piece;

  rule wiggle
  {
    shoot_piece --> left shoot_piece right shoot_piece
  }

  graphics
  {
    shoot_piece
    {
      move(0.5);
      color(0.7, 0.7, 0.7);
      cylinder(0.1, 1.0);
      move(0.5);
    }
    left { turn(-20); }
    right { turn(20); }
  }
}
```

In this example, we have introduced two new symbols, `left` and `right`, which are associated with turns to the left and to the right, respectively. The rule was also modified to actually use the new symbols. The graphical result is shown in Fig. 4.

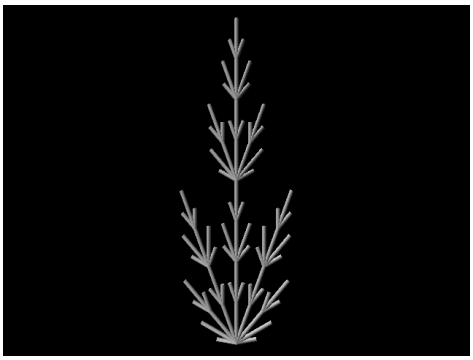

Figure 5: A branched structure generated with `L-transsys`.

You may be surprised about the irregular wiggling structure; it is due to the fact that `left` and `right` symbols occur in multiple repeats, sometimes cancelling each other out. If you're puzzled, the `ltrcheck` program might help you.

4.2.7 Push and Pop Symbols

Drawing branched structures requires functions for saving the current position and orientation, along with other information collecively called a state, and for returning to the saved state at some later time. These operations are called pushing (the state) and popping (the state), respectively. In `L-transsys`, pushing and popping are graphical functions, so in order to use these operations, we firstly have to define symbols for them and secondly, we have to associate them with the corresponding graphics primitives. Here's a code fragment that does all that:

```
symbol [;
symbol ];

graphics
{
  [ { push(); }
  ] { pop(); }
}

rule branch
{
  shoot_piece --> [ left shoot_piece ] [ right shoot_piece ]
```

```
                    shoot_piece shoot_piece
  }
}
```

This code shows that the square brackets, `[` and `]`, can be used as symbols. The intention is to use them as push and pop symbols, as shown here. In addition to introducing the new symbols and their graphics, the rule has again been modified to show off the new stuff. Fig. 5 shows that the system indeed yields a branched structure.

4.2.8 Integrating `transsys`

`transsys` is integrated into `L-transsys` by extending the concept of parametric L-systems: Symbols can be defined to have a `transsys` instance associated with them, as in

```
symbol meristem(cycler);
```

As you see, attaching a `transsys` instance to a symbol is really easy, the difficult part is to write the `transsys` program in the first place. For our example, we'll use the `cycler` program listed in section 4.1.7.

4.2.9 Handling `transsys` Instances in Rules

Having defined that `meristem` symbols should have a `cycler` instance attached, we now need a way to access what's going on in the `cycler` instance and to use this information for controlling the growth process modelled by `L-transsys` rules. Here's the code:

```
rule grow
{
  meristem(t) : t.A > 0.91 -->
       [ left meristem(transsys t: ) ]
       [ right meristem(transsys t: ) ]
       shoot_piece meristem(transsys t: )
}
```

This example introduces several new constructs. Firstly, we see that a `transsys` instance associated with a symbol is given a local name within the lefthand side

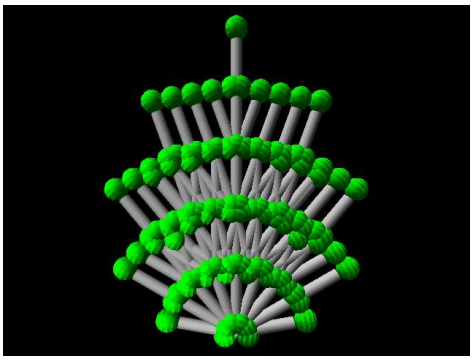

Figure 6: Another branched structure generated with `L-transsys`. In this case, branching is controlled by gene activity within `transsys` instances in the meristems which are graphically rendered as green spheres.

of a rule. Here, the `transsys` instance associated with the `meristem` symbol is labelled `t`.

Secondly, we see a new element in this rule: A condition which must be fulfilled in order to activate the rule. In our example, the rule is only activated if the concentration of factor `A` in the `transsys` instance labelled `t` is greater than 0.91.

Thirdly, the `transsys` instance is also used for creating new symbols on the righthand side of a rule. All `meristem` symbols specified there are qualified with `(transsys t: )`. This means that the factor concentration values from the `transsys` instance labelled `t` are copied into the `transsys` instances associated with the newly created symbols.

4.2.10 A Plant Branching Under the Control of Genes

Having introduced the handling of `transsys` instances in rules, we are finally ready to write an `L-transsys` program in which branching is controlled by gene activity within meristems. We just have to add the graphics instructions for rendering `meristem` symbols:

```
meristem
{
  move(0.2);
  color(0.0, 1.0, 0.0);
  sphere(0.2);
```

```
  move(0.2);
}
```

A graphical display of the virtual plant obtained with this `L-transsys` program is shown in Fig. 6.

4.2.11 Visualizing Factor Concentrations

In the `L-transsys` program which we've developed now, factor concentrations are causal for branching. So, it would be nice if we could somehow see the oscillations in concentrations of factor `A` as our plant develops. This is easily feasible, as factor concentrations are accessible within graphics definitions. All we need is to modify the meristem graphics:

```
meristem
{
  move(0.2);
  color(0.3 + A / 0.9 * 0.7, 0.3 + R / 20.0 * 0.7, 0.3);
  sphere(0.2);
  move(0.2);
}
```

This modification does not alter the morphology of the plant. But instead of showing all meristems as spheres with a bright green colour, the spheres are now coloured according to the concentrations of the two factors. This is best seen in an animation. Such an animation can be seen by running `ltransgl`: Firstly, let `ltransgl` do a few hundred derivations. Here, the fact that pressing `N` performs 50 derivations at once is handy. Once you've done that, press `Home` to get back to the axiom. Now press `>` to see an animation running through all steps. A reverse animation can be shown by pressing `<`.

5 Formal `transsys` Specification

5.1 Constitutents of a `transsys` program: Lexical Analysis

The `transsys` language consists of tokens (lexical elements), much as many common computer languages, such as C. The token classes are:

- Comment: All characters between a # (hash character) and the end of a line are considered a comment and are ignored by `transsys`.

- Number: All numbers in `transsys` are real valued and are written in the usual notation, e.g. `12`, `3.14`, `47.11e22` etc.

- Identifier: An identifier is a string in which the first character is an alphabetical character and the subsequent characters are alphanumerical characters. The underscore _ is considered an alphabetical character. Identifiers are case sensitive. Their main purpose in `transsys` is denoting genes and factors.

- Keyword: The keywords reserved by the core `transsys` language are `activate`, `constitutive`, `decay`, `default`, `diffusibility`, `factor`, `gauss`, `gene`, `product`, `promoter`, `random`, `repress`, `transsys`.

  Additionally, the following keywords are reserved by `L-transsys`: `axiom`, `bank`, `box`, `color`, `cylinder`, `graphics`, `lsys`, `move`, `pop`, `push`, `roll`, `rule`, `sphere`, `symbol`, `turn`.

  Reserved keywords may only be used in the ways defined by the language specifications, i.e. they cannot be used as identifiers or for other purposes. Please see section 5.7 below for additional info and advice.

- Operator: `transsys` uses the following operators, which should look familiar to those programming in C or C++: `<=`, `>=`, `==`, `!=`, `&&`, `x`, `+`, `-`, `*`, `/`, `!`, `<`, `>`, `=`.

  Additionally, `L-transsys` reserves and uses the production operator `-->`.

- Punctuation and Structure: Curly braces `{`, `}` are used to separate a `transsys` specification into modular blocks. Parentheses `(` and `)` are used for explicitly specifying precedence in arithmetic and logical expressions. Individual statements are separated by semicolons `;`, assignment lists in `L-transsys` are separated by commas.

5.2 Overall Structure

A `transsys` is specified by the keyword **`transsys`**, followed by a name and a body, consisting of `transsys` elements enclosed in curly braces. `transsys` elements are factor definitions and gene definitions.

An empty `transsys`, i.e. one with no factors or genes, is formally allowed but it may be of limited use.

```
transsys -> "transsys" identifier "{" transsys_element_list "}"
transsys_element_list -> /* empty */
transsys_element_list -> transsys_element_list factor_definition
transsys_element_list -> transsys_element_list gene_definition
```

5.3 Factor Definition

A factor in `transsys` is specified by the keyword **`factor`**, followed by a name and a body. Within the body, the factor's decay rate and diffusion rate are specified. These specifications may be omitted, in which case the default value of 0 is assumed for both parameters.

Formally, the grammar allows multiple specifications of both parameters; in the current parser implementation, the last specification will become effective. However, relying on this "feature" is strongly discouraged.

```
factor_definition -> "factor" identifier "{" factordef_components "}"
factordef_components -> /* empty */
factordef_components -> factordef_components factordef_component
factordef_component -> "decay" ":" expr ";"
factordef_component -> "diffusibility" ":" expr ";"
```

5.4 Gene Definition

A gene in `transsys` is specified by the keyword **`gene`**, followed by a name and a body. The contents of the body are subdivided into a promoter component and a product component. The promoter component specifies the level of the gene's expression activity as a function of the factor concentrations. The product component specifies which factor is encoded by the gene (i.e. which factor is synthesized upon the gene's expression). ene's activation).

```
gene_definition -> "gene" identifier "{" promoter_component product_component "}"
```

5.4.1 Promoter Definition

The promoter component specifies computation of the level of the gene's expression as a function of the factor concentrations. Currently, there are three types of statements possible in the **`promoter`** component. Each statement is evaluated to compute a contribution of activation (or repression). The activation contributed by statement i is denoted by a_i.

`constitutive` is the most generic type, this statement specifies an expression determining an amount of activation:

$$a_i = \text{ result of evaluating expression} \tag{1}$$

activate and **repress** statements are both preceded by a factor name f, and both have a list of two expressions as arguments. The arguments determine the specificity, denoted by a_{spec}, and the maximal rate of activation, denoted by $a_{\max}$. The actual amount of activation is calculated according to the Michaelis-Menten-equation:

$$a_i = \frac{a_{\max} c_f}{a_{\text{spec}} + c_f} \tag{2}$$

Repression is calculated by the same formula with the sign reversed:

$$a_i = -\frac{a_{\max} c_f}{a_{\text{spec}} + c_f} \tag{3}$$

Both parameters a_{spec} and $a_{\max}$ are specified by expressions, which allows modelling of modulation of activation by protein-protein interactions. The amount of product p synthesized through expression of gene g in a time step is given by

$$\Delta_g c_p = \begin{cases} a_{\text{total}} := \sum_i a_i & \text{if } a_{\text{total}} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

```
promoter_component -> "promoter" "{" promoter_statements "}"
promoter_statements -> promoter_statement
promoter_statements -> promoter_statements promoter_statement
promoter_statement -> "constitutive" ":" expr ";"
promoter_statement -> factor_combination ":" "activate" "(" expr "," expr ")" ";"
promoter_statement -> factor_combination ":" "repress" "(" expr "," expr ")" ";"
factor_combination -> identifier
factor_combination -> factor_combination "+" identifier
```

5.4.2 Gene Product Definition

```
product_component -> "product" "{" product_statements "}"
product_statements -> "default" ":" identifier ";"
```

5.5 Expressions

```
expr -> expr LOGICAL_OR and_expr
expr -> and_expr
and_expr -> and_expr LOGICAL_AND not_expr
and_expr -> not_expr
not_expr -> '!' not_expr
not_expr -> cmp_expr
cmp_expr -> cmp_expr '<' arithmetic_expr
```

```
cmp_expr -> cmp_expr '>' arithmetic_expr
cmp_expr -> cmp_expr LOWER_EQUAL arithmetic_expr
cmp_expr -> cmp_expr GREATER_EQUAL arithmetic_expr
cmp_expr -> cmp_expr EQUAL arithmetic_expr
cmp_expr -> cmp_expr UNEQUAL arithmetic_expr
cmp_expr -> arithmetic_expr
arithmetic_expr -> arithmetic_expr '+' term
arithmetic_expr -> arithmetic_expr '-' term
arithmetic_expr -> term
term -> term '*' value
term -> term '/' value
term -> value
value -> REALVALUE
value -> '(' expr ')'
value -> RANDOM '(' expr ',' expr ')'
value -> GAUSS '(' expr ',' expr ')'
value -> IDENTIFIER
value -> IDENTIFIER '.' IDENTIFIER
```

5.6 L-transsys

```
lsys -> LSYS_DEF IDENTIFIER @1 '{' lsys_element_list '}'
lsys_element_list -> lsys_element
lsys_element_list -> lsys_element_list lsys_element
lsys_element -> symbol_definition
lsys_element -> axiom_definition
lsys_element -> rule_definition
lsys_element -> graphics_definition
symbol_definition -> SYMBOL_DEF IDENTIFIER ';'
symbol_definition -> SYMBOL_DEF IDENTIFIER '(' IDENTIFIER ')' ';'
symbol_definition -> SYMBOL_DEF '[' ';'
symbol_definition -> SYMBOL_DEF '[' '(' IDENTIFIER ')' ';'
symbol_definition -> SYMBOL_DEF ']' ';'
symbol_definition -> SYMBOL_DEF ']' '(' IDENTIFIER ')' ';'
axiom_definition -> AXIOM_DEF production_element_string ';'
rule_definition -> RULE_DEF IDENTIFIER '{' rule_components '}'
rule_components -> rule_lhs ':' expr ARROW rule_rhs
rule_components -> rule_lhs ARROW rule_rhs
rule_lhs -> lhs_element_string
lhs_element_string ->           /* empty */
lhs_element_string -> lhs_element_string lhs_element
lhs_element -> IDENTIFIER
lhs_element -> IDENTIFIER '(' IDENTIFIER ')'
rule_rhs -> production_element_string
production_element_string ->            /* empty */
```

```
production_element_string -> production_element_string production_element
production_element -> IDENTIFIER
production_element -> IDENTIFIER '(' transsys_initializer ')'
production_element -> '['
production_element -> ']'
transsys_initializer -> source_transsys_specifier assignment_list
transsys_initializer -> assignment_list
source_transsys_specifier -> TRANSSYS_DEF IDENTIFIER ':'
assignment_list ->              /* empty */
assignment_list -> assignment
assignment_list -> assignment_list ',' assignment
assignment -> IDENTIFIER '=' expr
graphics_definition -> GRAPHICS_DEF '{' symgraph_list '}'
symgraph_list ->                /* empty */
symgraph_list -> symgraph_list symgraph
symgraph -> IDENTIFIER '{' graphcmd_list '}'
symgraph -> '[' '{' graphcmd_list '}'
symgraph -> ']' '{' graphcmd_list '}'
graphcmd_list ->                /* empty */
graphcmd_list -> graphcmd_list graphcmd
graphcmd -> MOVE '(' expr ')' ';'
graphcmd -> PUSH '(' ')' ';'
graphcmd -> POP '(' ')' ';'
graphcmd -> TURN '(' expr ')' ';'
graphcmd -> ROLL '(' expr ')' ';'
graphcmd -> BANK '(' expr ')' ';'
graphcmd -> SPHERE '(' expr ')' ';'
graphcmd -> CYLINDER '(' expr ',' expr ')' ';'
graphcmd -> BOX '(' expr ',' expr ',' expr ')' ';'
graphcmd -> COLOR '(' expr ',' expr ',' expr ')' ';'
```

5.7 Future Development Plans

5.7.1 Additional Keywords

It is a notorious problem that the set of keywords of a computer language increases as the language evolves, and that the introduction of new keywords may result in old code becoming malfunctional and "illegal".

Extending `transsys`, particularly by integration of additional modelling methods, is definitely intended. However, it is not clear which keywords will be introduced in this process. Therefore, at this time, only some rather general design principles can be given. These may provide some guidance for the cautiously minded:

- No keywords containing any capital letters will be introduced.
- No keywords beginning with an underscore will be introduced, and introduction of keywords containing underscores is unlikely.
- Introduction of keywords containing numbers is also unlikely
- Most likely candidates for keywords are on the one hand words that are used as keywords in other computer languages, an on the other hand terms for generic biological structures. Examples for the latter category are **`cell`**, **`tissue`**, **`organ`**, **`enhancer`**, **`chromosome`** etc. It is recommended to avoid such terms in order to minimize troubles due to future development of `transsys`.