

SKETCHBOOK Manual

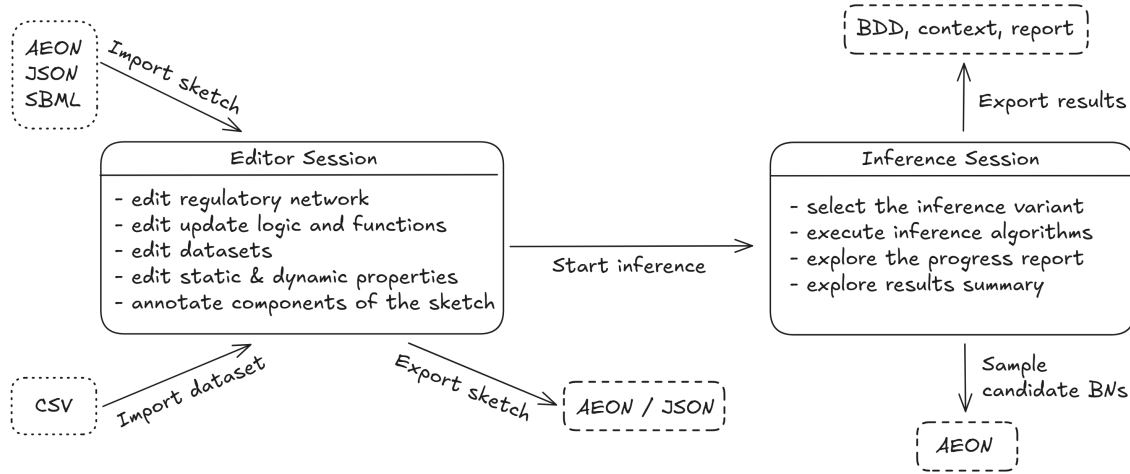
Contents

1	Running SKETCHBOOK	3
2	Sketch editor session	4
2.1	General navigation	4
2.2	Editing regulatory network	4
2.3	Editing update functions	6
2.4	Loading and editing datasets	8
2.5	Editing properties	9
2.6	Displaying annotations list	12
2.7	Starting analysis and checking consistency	13
3	Inference session	15
4	Sketch format	18
4.1	Logical expressions	18
4.1.1	Syntax of update functions	18
4.1.2	Syntax of generic dynamic properties	18
4.1.3	Syntax of generic static properties	19
4.2	Datasets	19
4.3	Sketch formats	20
4.3.1	Custom JSON format	20
4.3.2	Extended AEON format	20
4.3.3	SBML format	20

Intro to SKETCHBOOK

SKETCHBOOK is a tool for inference of Boolean networks (BNs) using the framework of Boolean network sketches [3]. BN sketch is a concept for combining several types and sources of knowledge with experimental data. It allows us to apply inference algorithms to find all admissible BNs. Particularly, a BN sketch consists of an influence graph, a partially specified Boolean network (PSBN), and sets of required static and dynamic properties of the model.

Specifically, SKETCHBOOK is a multiplatform desktop application that offers a graphical interface for both editing all components of the sketch and running the inference process. SKETCHBOOK consists of two main sessions or windows: The interactive *editor session* and an *inference session*, as illustrated by the workflow below.



General workflow of SKETCHBOOK.

In the editor window, you can edit PSBNs, load datasets with observations, and create various kinds of static and dynamic properties. Once you have created the sketch, you can open the inference window. You can then run the inference algorithms and explore the set of admissible BNs that satisfy all the properties.

This manual covers the following:

- We give short instructions on the setup of SKETCHBOOK in Section 1.
- We describe the sketch editor session of the tool and its features in Section 2.
- Inference session and results are described in Section 3.
- Supported formats for datasets, sketches, and syntax for various expressions are discussed in Section 4.

1 Running SKETCHBOOK

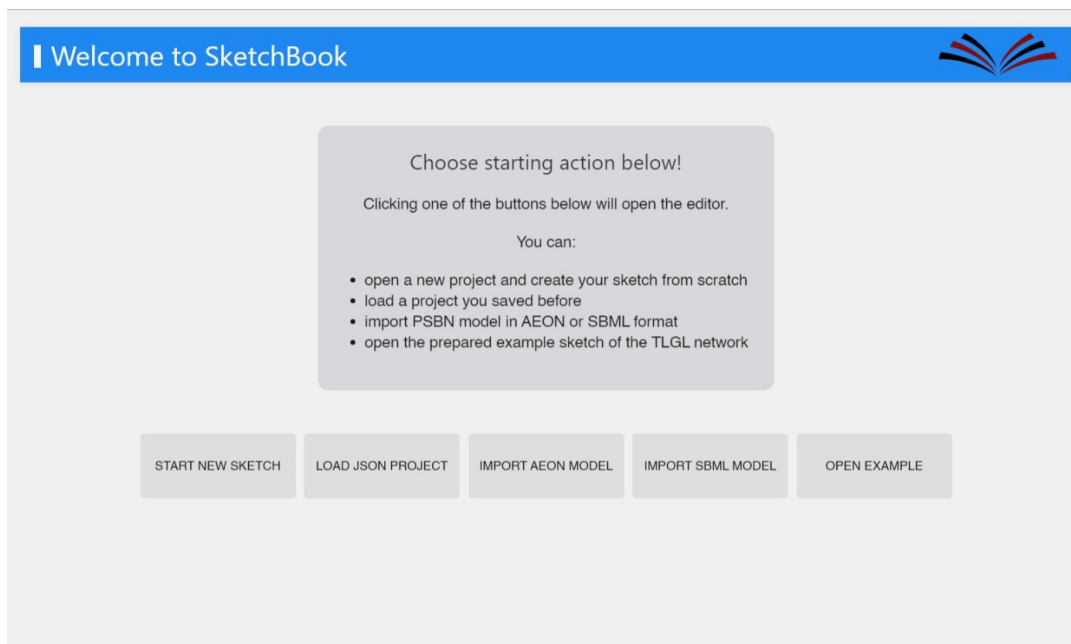
This section provides general instructions for installing and setting up SKETCHBOOK. The tool's repository with the latest version of the source code (that is licensed under MIT license) is freely available at our GitHub: <https://github.com/sybila/biodivine-sketchbook>.

We provide pre-built binaries and installation files for the application in the [release section](#) of the repository. To start using Sketchbook, choose the latest release and download a binary for your operating system. You choose between `.app` and `.dmg` for **macOS**, `.AppImage`, `.deb` and `.rpm` for **Linux**, or `.exe` and `.msi` for **Windows**. If you need a different pre-built binary for a specific platform, let us know!

Note that the binaries are not signed with official developer certificates, so macOS and Windows will most likely require you to grant special permissions to run the app. On newer versions of macOS, the message is that the app is *corrupted*. This is still the same issue regarding app certificates. You should be able to “enable” the app by running `xattr -c /path/to/biodivine_sketchbook.app`.

Alternatively, the desktop application can also be built directly from the repository's source code. To do this, please consult the *Development guide* of the repository Readme. Note that the local build requires additional dependencies to be installed.

After successfully starting the application, you should see the following screen:



Initial screen of SKETCHBOOK.

2 Sketch editor session

The sketch editor is the primary session of the tool that allows to edit all components of the sketch. The tool starts with a simple initial screen where you can select first action. This can be loading a sketch in various formats, creating a new empty sketch, or choosing a prepared example. Choose one of the buttons and continue:

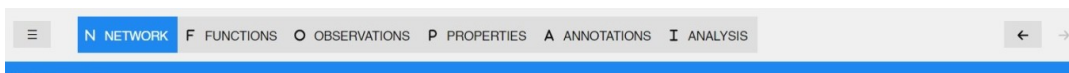


Available buttons at the initial screen.

The following subsections then summarize the options available in the editor.

2.1 General navigation

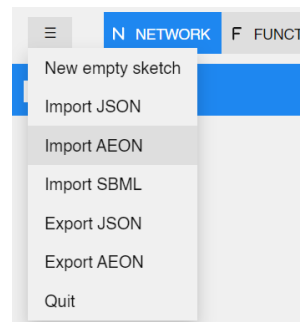
At the top of the screen, there is a popup menu, a list of tabs, and an undo-redo button.



Navigation bar at the top of the screen.

Each tab offers different functionality, as discussed below. You can select any of them by clicking on the tab button. Tabs can also be "locked," allowing two tabs to be displayed side by side. To do so, click on the small "lock button" at the bottom left of the screen.

The menu offers options regarding import/export. You can import sketches in JSON, AEON, or SBML format, and we also offer export in JSON and AEON. More details on the formats is in Section 4. Note that importing new sketch results in losing current data (you'll get a warning).

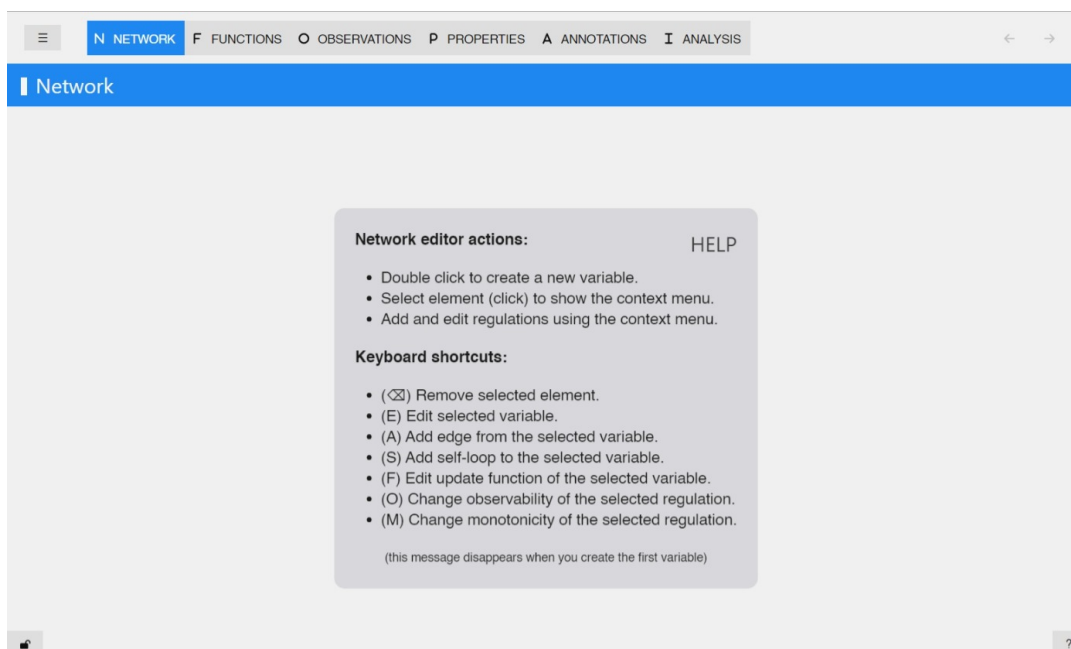


Menu with all the options.

Lastly, you can use the undo and redo buttons in the top right corner to reset or reapply your last actions.

2.2 Editing regulatory network

The editor starts with the *Network* tab opened. If a new project is started, a help message is displayed. The message disappears after you start editing the network, but you can reopen it by hovering over the "question mark" button in the lower-right corner of the screen. The user can then edit or load the network.



Empty network tab of the editor session with a help message.

A new variable can be added by double-clicking at the free space. You can drag variables to achieve any kind of layout. Then, you can select a variable to get a context menu with options to add regulations, edit the variable, or edit its update function. Regulation can be added by clicking the corresponding “+” button and dragging the arrow that appears towards another variable. The option for editing a variable opens a dialogue window where you can edit its name, ID, and annotation.

Example of a dialogue window to edit variable.

You can also select a regulation and customize it. We offer customizing two kinds of regulatory properties – *monotonicity* and *observability*.

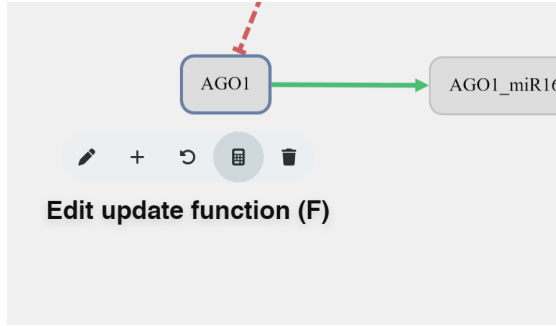
Monotonicity allows us to specify what type of effect the interaction has. The monotonicity of the regulation is reflected by its colour. The options are:

- **activation** for positive monotonicity (green colour)
- **inhibition** for negative monotonicity (red colour)

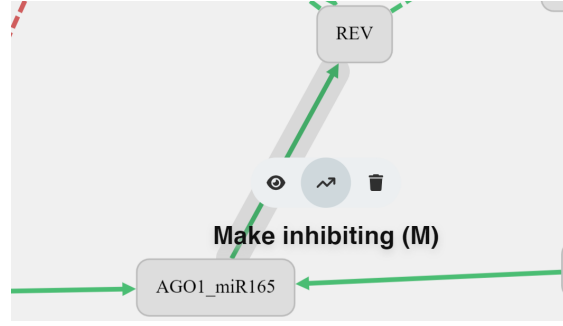
- **dual** for non-monotonous effect (blue colour)
- **unknown** for undefined (grey color)

Essentiality allows to specify whether the regulation must always have an effect on its target, or whether it may be “redundant”. Essentiality is reflected by different line styles. The options are:

- **essential** for regulations that must always have an effect (solid line)
- **unknown** regulations that may or may not have an effect (dashed line)
- **non-essential** for regulation being completely redundant (dotted line)

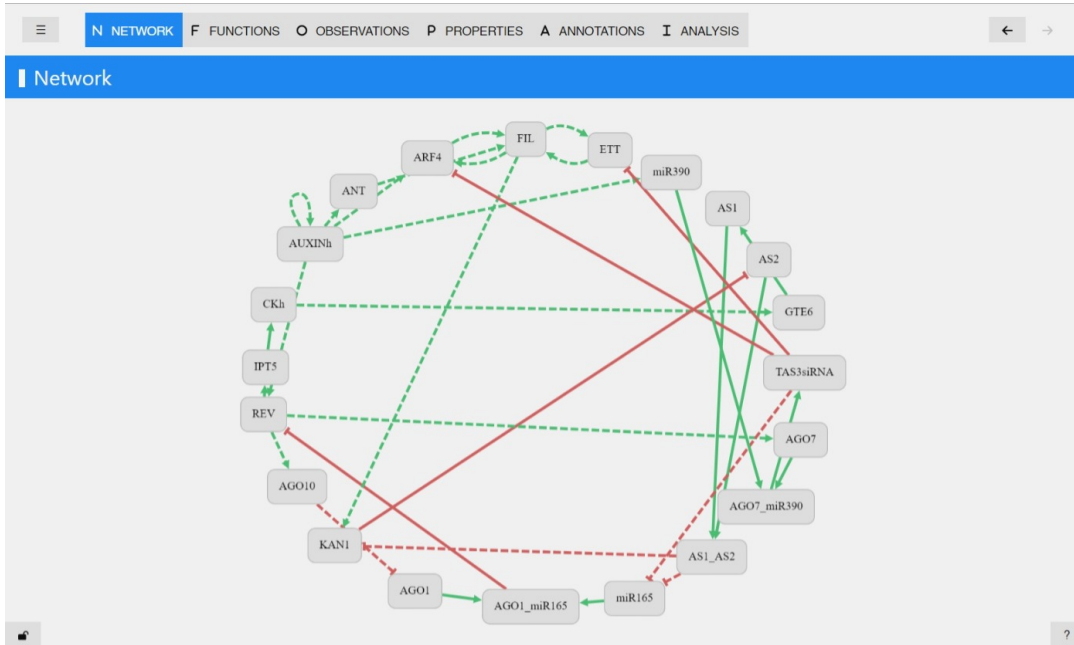


Variable context menu.



Regulation context menu.

The fully edited network can look like the following:

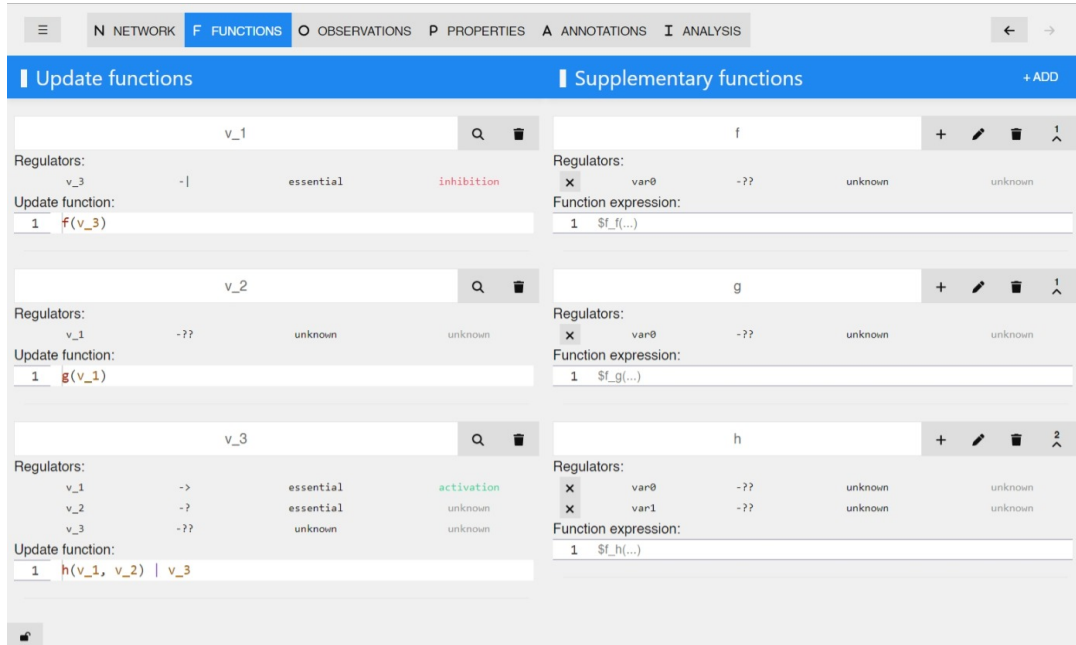


Network tab with an example of a created regulation network.

2.3 Editing update functions

Update rules and supplementary functions can be edited within the *Functions* tab. On the left side of the screen, you can see a list of variables, their regulators and their *update functions*. On the

right side, you can create additional *supplementary functions* that can be used inside the update function expressions.



Functions tab with partially specified update functions for three variables and three supplementary functions.

For each variable, you can edit the type of regulations by clicking on the monotonicity or essentiality value. This is propagated to the network tab. You can also edit the update function's expression. The format of update functions is discussed in Section 4.1.1.



List of regulators and update expression for variable v_3 .

You can add new supplementary functions at the top bar by clicking the +ADD button. You must always add a new supplementary function before using it in any expressions. For each function, you can increment its arity, edit its details (same as for variable, via an external dialogue), or delete it. Arguments of the function can be deleted to decrement its arity. The details can be hidden by collapsing the function.

g

h

Regulators:

X	var0	-??	unknown	unknown
X	var1	-??	unknown	unknown

Function expression:

1 \$f_h(...)

Collapsed function g with arity 1 and expanded h with arity 2.

2.4 Loading and editing datasets

Datasets of observations can be edited in the *Observations* tab.

Observations

+ CREATE + IMPORT

d1 (Dataset 1)

EDIT DATASET + ADD ROW + ADD COLUMN DELETE DATASET

Index	Name	ID	AGO1	AGO10	AGO7	ANT	ARF4	AS1	AS2	ETT	FIL	KAN1	miR165	miR390	REV
1	Observation1	Observation1	1	0	0	1	1	0	0	1	1	1	1	1	0
2	Observation2	Observation2	0	1	1	1	0	1	1	0	0	0	0	1	1

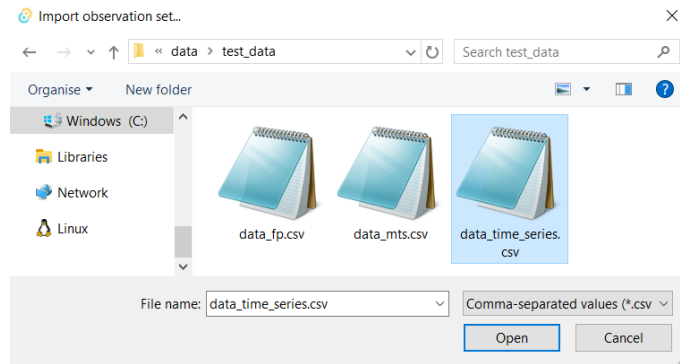
Page Size 20 First Prev 1 Next Last

d2 (Another dataset)

dataset_3 (dataset_3)

Observations tab with three datasets, one of them being expanded.

You can either import a CSV dataset by clicking on the +IMPORT button (in the format discussed in Section 4) or create one from scratch (clicking on the +CREATE button). The import button opens a file dialogue and you can choose the CSV with the data. You can expand the dataset to see its content in a table-based editor (by clicking “+” button next to its ID) or keep it collapsed.



Dataset import dialogue.

The table-based editor allows editing of all aspects of the dataset. The dataset's metadata (name, ID, and variable names) can be edited by selecting the **EDIT DATASET** option. You can also add new rows (observations) and columns (variables). Values of individual observations can be edited by clicking on the table's fields or by selecting the blue edit button. You can delete both observations or datasets by clicking the corresponding pink **DELETE** buttons.

☐ d1 (Dataset XYZ)

☒ EDIT DATASET
 + ADD ROW
 + ADD COLUMN
 ☐ DELETE DATASET

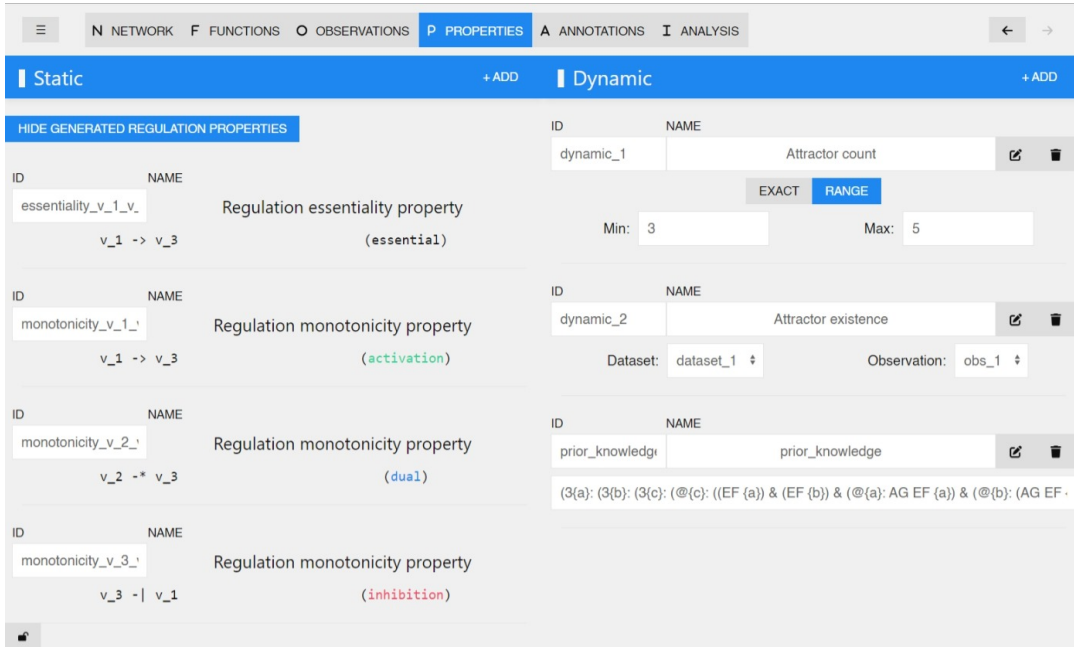
<input type="checkbox"/>	Index	Name	ID	var_A	var_B	var_C	var_D	var_E	var_F	var_G		
<input type="checkbox"/>	1	measurement XYZ	obs1	1	1	1	1	1	1	1	Edit	Delete
<input type="checkbox"/>	2	measurement 123	obs2	0	0	0	0	0	0	0	Edit	Delete
<input type="checkbox"/>	3	measurement ABC	obs3	1	0		1	0			Edit	Delete

Page Size 20
First Prev 1 Next Last

Zoom to a table-based editor for a dataset.

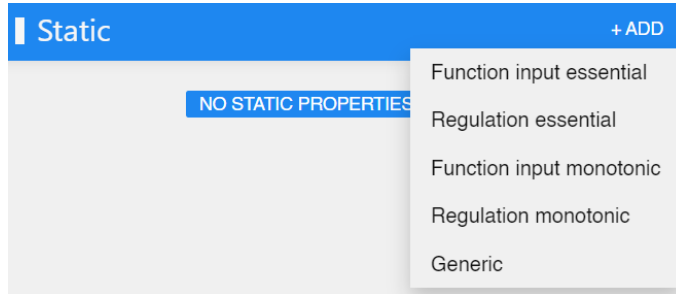
2.5 Editing properties

Properties tab serves to create and edit properties. It is divided into static and dynamic properties. Static properties place requirements on the structure of the model (on its update functions and regulations) while dynamic properties specify how the model must behave.



Properties tab with four static properties and three dynamic properties.

For static properties, you can write a **generic** property in first-order logic (FOL); the syntax is described in Section 4.1.3). We also offer templates to specify *monotonicity* and *essentiality* of regulations and supplementary functions. For monotonicity properties, the user chooses from the following options – *activation*, *inhibition*, *dual* (non-monotonic), or *unknown* (can be any). For essentiality properties, the options are: *essential*, *non-essential* (has no effect), and *unknown* (could be any). Some properties are automatically derived from the regulation constraints selected in the network tab. The user can hide these properties with the button at the top, as there may be numerous. All user-created properties can also include a FOL context formula. The properties with such context are required to hold only in situations when the context formula is satisfied.



Template selection for static properties.

ID	NAME
static_1	var_1 inhibits var_2

var_2

|-

var_1

(inhibition)

Context formula:

true

Example of a regulation monotonicity property.

For dynamic properties, you can write a generic property in hybrid CTL (HCTL); the syntax is described in Section 4.1.2). We also offer a list of predefined templates for dynamic properties. There are templates for the following cases:

- *Exists trap space*: selected observation corresponds to a trap space
- *Exists fixed point*: selected observation corresponds to a fixed point
- *Exists attractor*: selected observation corresponds to an attractor
- *Exists trajectory*: selected dataset corresponds to a time-series trajectory
- *Attractor count*: model admits selected number (range) of attractors

Dynamic
+ ADD

NO DYNAMIC PROPERTIES DEFINED

+ ADD

Exists trap space

Exists fixed point

Exists trajectory

Attractor count

Exists attractor

Generic

Template selection for dynamic properties.

For the templates regarding the existence of attractors, trap spaces, or fixed points, you can select a dataset and its observation. This observation is used to automatically generate an HCTL formula stating: “There exists a state corresponding to the given observation that lies in an attractor (or a trap space, or fixed point).” If the observation is omitted, the formula is created for all observations in the dataset. The *trap space* property also lets you specify whether the trap space should be minimal or non-percolable.



ID	NAME
dynamic_2	Fixed point at dataset d_1 observation obs_1

Dataset: d_1

Observation: obs_1

Example of a fixed-point property.

The *trajectory* property encodes time series by requiring the existence of a trajectory between states corresponding to subsequent observations in a dataset. For the *attractor count* property, the user can specify how many attractors the model should admit.

ID	NAME	
dynamic_1	less than 3 attractors	 

EXACT RANGE

Min:

Max:



Example of an attractor count property.

2.6 Displaying annotations list

Throughout the various tabs, you can create string annotations for entities such as variables, functions, datasets, and properties. Use the edit button option at the corresponding entity to open an edit dialogue and set the annotation.

Static + ADD

HIDE GENERATED REGULATION PROPERTIES

ID	NAME	
static_1	var_1 inhibits var_2	 

var_2

| -

var_1

(inhibition)

Context formula:

Edit property (static_1 / var_1 inhibits var_2)

ID

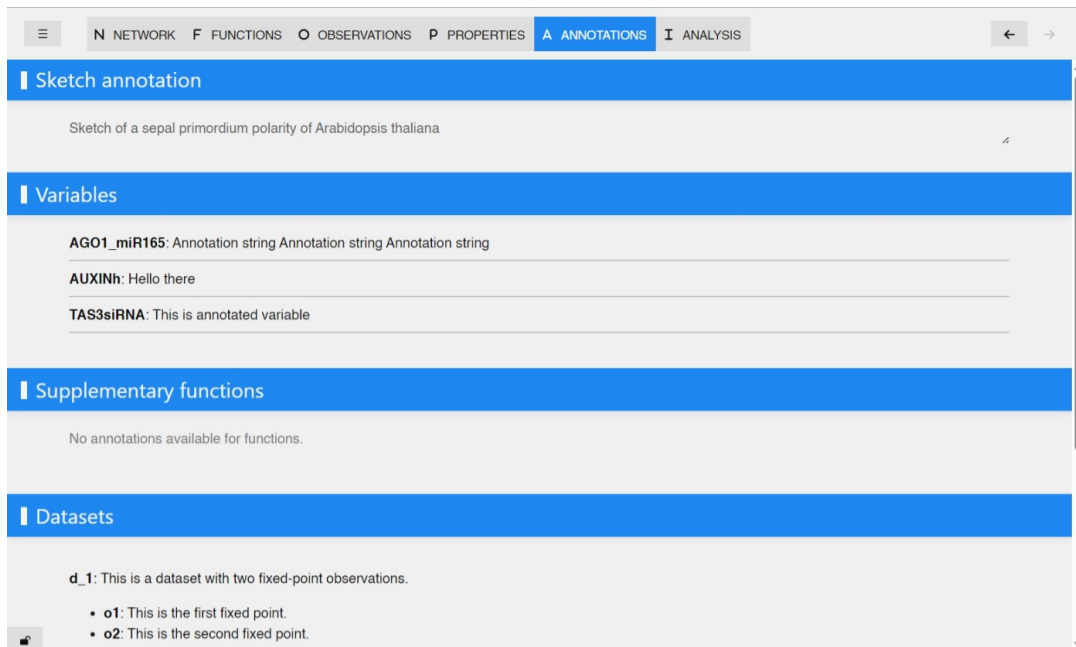
NAME

ANNOTATION

SAVE

Example of an edit dialogue for a static property.

All of the annotations created throughout the sketch are summarized in the *Annotations* tab. Annotations are divided according to the entity type. For each entity, its ID and the annotation are shown. Datasets are shown together with their observations.



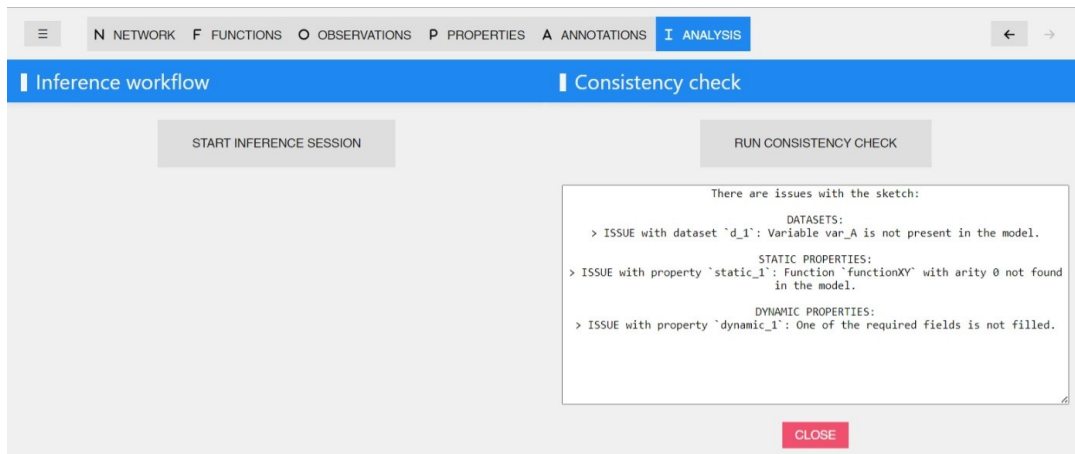
Annotations tab with annotations divided by entity types.

2.7 Starting analysis and checking consistency

The last part of the editor session is the *Analysis* tab. The current state is very simple; you can start the inference session or explicitly check the consistency of the sketch.

After clicking on the **START INFERENCE SESSION**, a new window opens. We go into detail on the inference session functionality in Section 3. You can open multiple inference sessions at the same time. After opening a new session, it uses the version of the sketch at the time of its creation. Subsequent changes made in the editor do not affect any already running inference.

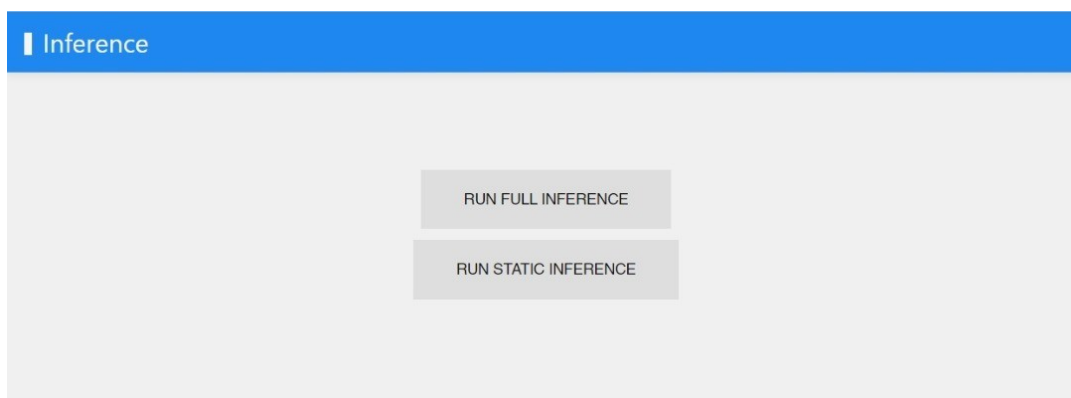
Most of the user inputs are immediately automatically validated. If an issue is encountered, the corresponding action should not be taken, and the user should be shown an error message. However, there can still be some inconsistencies in the sketch – property templates with some fields not filled, logical formulas referencing non-existent variables, and so on. To check the consistency of the sketch, click on the **RUN CONSISTENCY CHECK**. An exhaustive static check is run on the sketch, reporting all issues in the text area under the button.



Analysis tab with a button to start inference window and run consistency check. An example of inconsistencies is shown in the text window.

3 Inference session

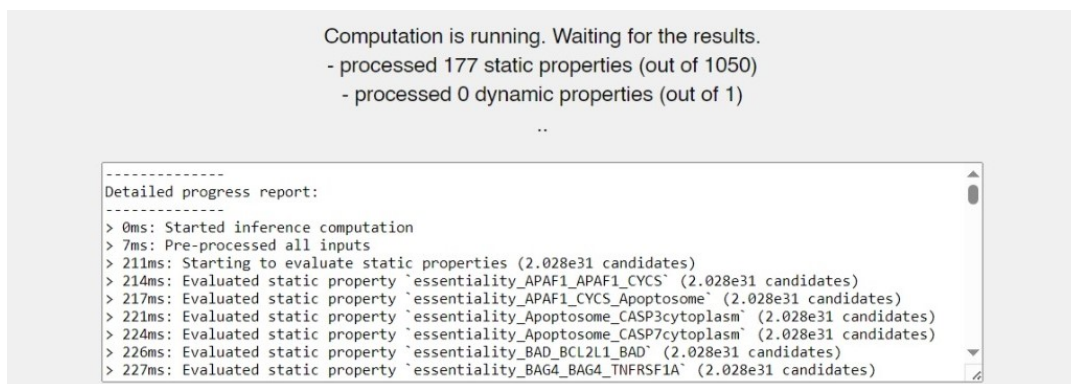
After starting the new inference session, a simple window appears where you can run the inference. You can choose from two options: running the full inference with all properties or performing a preliminary step – a partial “static” inference with static properties only. The static properties are usually much faster to evaluate. You can check the results, and if the candidate set is too large, you might first want to add more static properties, before running the full inference.



Initial inference screen with the options to run two variants of inference.

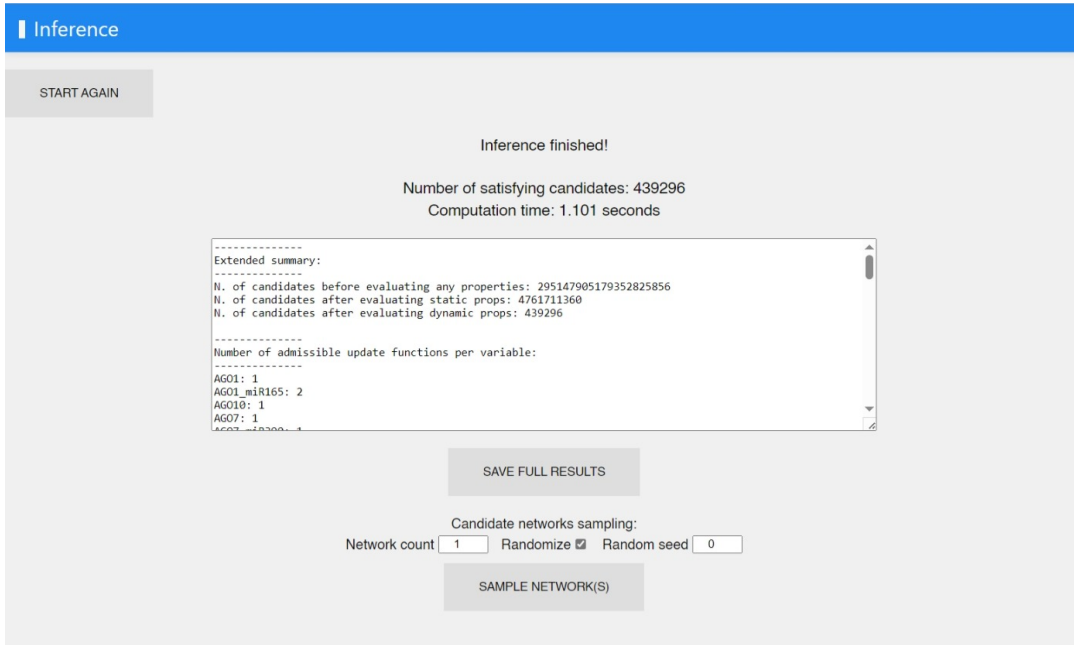
Once you click one of the buttons to run the inference, the backend starts the computation. You’ll see the progress summary and a report. In the report, you’ll see various checkpoints reached during the computation, such as when all inputs are processed or when a property is evaluated. For each such checkpoint, the time of reaching it and the number of remaining consistent candidates are reported.

At any point, you can cancel and restart the computation by clicking the **START AGAIN** button. In the background, the ongoing computation completes its current task (such as evaluating a single property) and gets terminated.



An example of the intermediate progress output during computation.

Once the computation finishes, you’ll see a screen with a summary of results, a progress report, and options to export the results or sample admissible BNs.



Inference window with a summary and buttons to export/sample the results.

Summary and report The summary includes the number of admissible candidates at various stages of the algorithm, computation times, and the number of admissible update functions per variable. The summary of admissible update functions can help to identify which parts of the model are consistent across candidates and which are more variable, enabling refining the sketch accordingly. For example, if you see that there is only one option for an update function possible for a given variable, you can automatically refine your sketch before running subsequent computations.

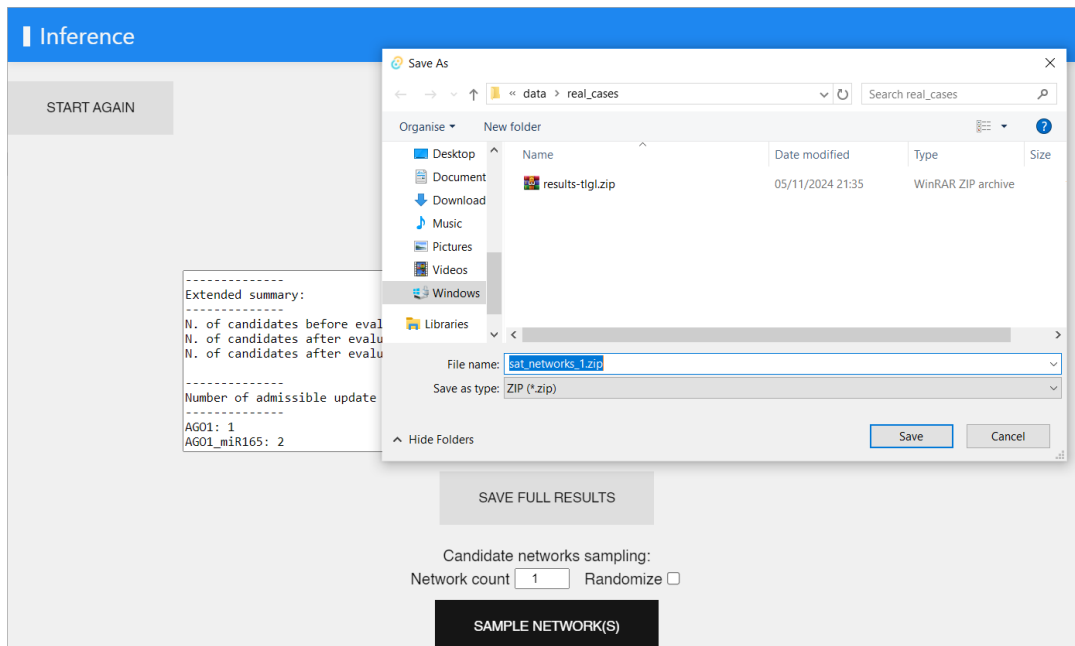
Export of results You can export the full results (by clicking the **SAVE FULL RESULTS**) button. After you choose the path, the results are exported in an archive that contains the following components:

- A serialized BDD encoding all the admissible BN interpretations. The BDD format is compatible with the format used by the `aeon.py` library [1].
- Original sketch used as an input for the inference. This is included for replicability.
- Derived parametrized `aeon` model used internally during the computation. This model gives context to the BDD variables.
- Computation report encompassing all the information shown in the UI.
- Folder with a list of admissible update function variants per each variable.

Name	Size
..	
admissible_update_functions	25
color_bdd.bdd	49
derived_model.aeon	71
original_sketch.json	5,031
report.txt	1,401

Results archive structure.

Sampling admissible networks Finally, you can randomly sample the fully specified admissible candidate networks (as `.aeon` files). To generate witness networks, click on the **SAMPLE NETWORK(S)** button. You can select how many witnesses to generate and also select a random sampling (the default sampling is deterministic). By clicking the button, a file save dialogue will open. Witness files are saved in a `.zip` archive.



Example of a sampling dialogue.

4 Sketch format

In the first part of this section, we describe our syntax for various logical expressions that users may utilize when writing update functions or properties from scratch. We then continue by describing the formats that the tool supports for the import/export of datasets and the whole sketch.

Before going into details, let's discuss some general rules regarding identifiers. Various entities (variables, functions, ...) of the sketch can have IDs, names, and annotations. An ID is a unique combination of alphanumeric characters and underscores; an ID should not start with a number. The name is an arbitrary combination of characters; we only require it not to contain newlines. Annotation is a completely arbitrary string.

4.1 Logical expressions

4.1.1 Syntax of update functions

Each variable has an update function that dictates its behaviour. Update functions are logical expressions in the format given by the following grammar. Note that *varId* is an ID of some variable and *fnId* is an ID of a function.

$$\begin{aligned} \text{function} &= (\text{function}) \mid \text{const} \mid \text{varId} \mid \text{fnSymbol} \mid !\text{function} \mid \text{function OP function} \\ \text{const} &= \text{true} \mid \text{false} \\ \text{OP} &= \& \mid \mid \mid \Rightarrow \mid \Leftrightarrow \\ \text{fnSymbol} &= \text{fnId}(\text{args}) \\ \text{args} &= \text{function} \mid \text{args}, \text{args} \end{aligned}$$

For example, the update function $C \rightarrow (B \wedge \mathbf{h}(A, C))$ could be written as:

$$C \Rightarrow (B \ \& \ \mathbf{h}(A, C))$$

4.1.2 Syntax of generic dynamic properties

Apart from creating dynamic properties using pre-defined templates, users can define generic properties using logic HCTL.

The following grammar defines our textual format for HCTL formulae:

$$\begin{aligned} \text{formula} &= (\text{formula}) \mid \text{const} \mid \text{propName} \mid \text{var} \mid \text{unaryOp formula} \\ &\quad \mid \text{formula binaryOp formula} \\ &\quad \mid \text{hybridOp var} : \text{formula} \\ \text{const} &= \text{true} \mid \text{false} \\ \text{var} &= \{ \text{varName} \} \\ \text{unaryOp} &= \sim \mid \mathbf{AX} \mid \mathbf{EX} \mid \mathbf{AF} \mid \mathbf{EF} \mid \mathbf{AG} \mid \mathbf{EG} \\ \text{binaryOp} &= \& \mid \mid \mid \Rightarrow \mid \Leftrightarrow \mid \sim \mid \mathbf{AU} \mid \mathbf{EU} \\ \text{hybridOp} &= ! \mid 3 \mid \mathbf{V} \mid @ \mid \backslash \text{bind} \mid \backslash \text{exists} \mid \backslash \text{forall} \mid \backslash \text{jump} \end{aligned}$$

Note that *varName* stands for an HCTL variable, while *propName* stands for an atomic proposition. Each *propName* must correspond to a valid BN variable ID (this is checked). Only alphanumeric characters and underscores can appear in *varName* and *propName*. Note that \sim corresponds to the negation and \sim to the xor operator. For hybrid operators, we allow specifying them by name prefixed by a backlash or using the following symbols: $!$ for the bind operator, 3 for the existential

quantifier, \forall for the universal quantifier, and $@$ for the jump operator. Correspondence with other logical and temporal operators is straightforward.

The operator precedence is the following (the lower, the stronger):

- unary operators (both negation and temporal): 1
- binary temporal operators: 2
- Boolean binary operators: and=3, xor=4, or=5, imp=6, eq=7
- hybrid operators: 8

However, it is strongly recommended to use parentheses wherever possible. Note that formulae must not contain free variables.

For example, formula $\exists x. \forall y (@_y. \mathbf{EF}(x \wedge (\downarrow z. \mathbf{AX} z)))$ could be written as:

```
\exists {x}: \forall {y}: (\jump {y}: EF ({x} & (\bind {z}: AX {z})))
```

4.1.3 Syntax of generic static properties

Similarly, you can also design your own generic static properties. For these, we use the following first-order logic (FOL) syntax.

```
formula = ( formula ) | const | var | fnSymbol | !formula
          | formula binaryOp formula
          | quantifier varList : formula
const = true | false
binaryOp = & | | => | <=> | ^
fnSymbol = fnId(args)
args = function | args, args
quantifier = 3 | V | \exists | \forall
varList = var | var, varList
```

Note that *var* stands for a FOL variable, and *fnId* is an ID of a function. Only alphanumeric characters and underscores can appear in *var* and *fnId*. For quantifiers, we allow specifying them by name prefixed by a backlash or using the following symbols: 3 for the existential quantifier, and V for the universal quantifier. This is the same as for the HCTL syntax.

For example, the monotonicity formula $\forall x, y. f(x, y, 1) \rightarrow f(x, y, 0)$ could be written as:

```
\forall x, y: f(x, y, 1) => f(x, y, 0)
```

4.2 Datasets

In the observations tab of the tool's editor, users can load binarized datasets from a CSV file. The first line of the CSV describes the dataset's variables, while the remaining lines describe individual observations. Let's look into an example:

```
ID, A, B, C
o1, 1, 0, 1
o2, 0, *, 0
```

This dataset contains two observations *o1* and *o2* over three variables *A*, *B*, and *C*. Values can be binary *1* or *0*, or a *** character for unknown value.

4.3 Sketch formats

SKETCHBOOK currently supports three formats for importing or exporting data – a fully custom JSON format, a novel extension of AEON format, and an SBML format. For both JSON and AEON, we support both the export and import of the whole sketch. We provide many examples of both formats in our repository. For the SBML format, we allow importing standard PSBN models in the SBML-qual format (there is no standardized extension to cover properties or datasets at the moment).

4.3.1 Custom JSON format

Our custom JSON-based format reflects a simplified version of the internal data structure. We can both import and export the whole sketch in this format. This format is used for fast and simple serialization and deserialization. We recommend using it unless you need to use the same model for different tools. For that, we recommend using the AEON format.

4.3.2 Extended AEON format

The original AEON format allowed specifying variables (simple IDs), regulations, functions, and layout. More recently, it was extended with custom model annotations that can be used to represent additional information in a structured way. This format is still compatible with all tools using the AEON format, as these tools simply ignore the additional annotation lines. For example, you can directly import models created in AEON [2], and the other way around.

The influence graph of the PSBN is described as a list of edges, where each edge is encoded as:

regulator edge_type target

Here, **regulator** and **target** are the names of the network variables, and the **edge_type** is the type of influence, corresponding to one of $\{->, -|, -?, ->?, -|?, -??\}$. Arrow $->$ denotes activation, $-|$ inhibition and $-?$ is for unspecified monotonicity. Finally, an extra $?$ signifies a non-observable regulation (a regulation that may or may not affect the target).

A particular update function for variable **A** is written as: “**\$A: function**”. The syntax of update functions is discussed in Section 4.1.1.

The layout position of variable nodes is written as “**#position:VAR:X,Y**”, where **VAR** is the variable’s ID, and **X**, **Y** are float positions (such as 42.42).

Our extension of the AEON format uses the annotations to represent all the additional information contained in the sketch. Each AEON annotation consists of a unique path-like identifier and a value. In our case, the path is simple – an entity type and ID. Note that IDs must always be unique in the sketch. Entity types can be *variable*, *function*, *static_property*, *dynamic_property*, or *dataset*. The value is a JSON-serialized struct of a given type (similar to our custom JSON format).

4.3.3 SBML format

The [SBML-qual](#) format can be used for importing standard PSBN models, without properties or datasets. However, you can add these in SKETCHBOOK, and export the resulting sketch in any other format. For more details, you can see the details regarding the syntax rules, you can read on [syntax for the SBML-qual format](#) as supported by the AEON tool [2]. We currently use a similar parser here.

References

- [1] Beneš, N., Brim, L., Huvar, O., Pastva, S., Šafránek, D., Šmijáková, E.: AEON.py: Python library for attractor analysis in asynchronous Boolean networks. *Bioinformatics* **38**(21), 4978–4980 (2022). <https://doi.org/10.1093/bioinformatics/btac624>
- [2] Beneš, N., Brim, L., Kadlec, J., Pastva, S., Šafránek, D.: AEON: attractor bifurcation analysis of parametrised boolean networks. In: *Computer Aided Verification*. vol. 12224, pp. 569–581. Springer (2020)
- [3] Beneš, N., Brim, L., Huvar, O., Pastva, S., Šafránek, D.: Boolean network sketches: a unifying framework for logical model inference. *Bioinformatics* **39**(4) (04 2023)