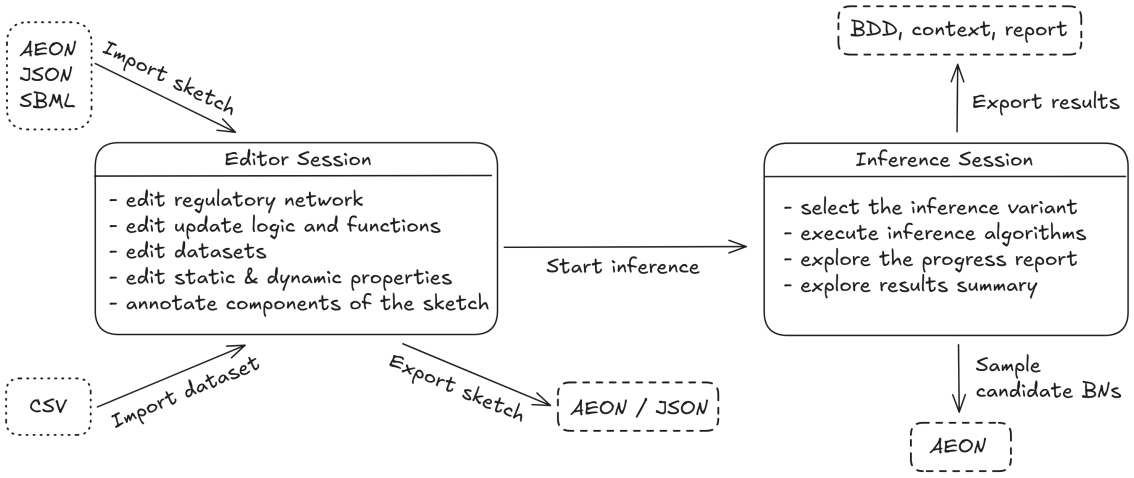
Sketchbook Manual

Contents

Intro to Sketchbook

Sketchbook is a tool for inference of Boolean networks (BNs) using the logic-based framework of BN sketches [3]. BN sketch provides a formal model specification that integrates diverse types of prior knowledge and experimental data, enabling the application of inference algorithms to identify all admissible BNs. Particularly, a BN sketch consists of an influence graph, a partially specified Boolean network (PSBN), and sets of required structural and dynamic properties that the model must satisfy.

Specifically, Sketchbook is a multiplatform desktop application that offers a graphical interface for both editing all components of the BN 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.


AEON
JSON
SBML
Import sketch
Editor Session
- edit regulatory network
- edit update logic and functions
- edit datasets
- edit static & dynamic properties
- annotate components of the sketch
CSV
Import dataset
Export sketch
AEON / JSON
Start inference
Inference Session
- select the inference variant
- execute inference algorithms
- explore the progress report
- explore results summary
Export results
BDD, context, report
Sample candidate BNs
AEON

General workflow of Sketchbook.

In the editor window, you can create and validate the model specification (i.e., design the Boolean network sketch) by editing the influence network, PSBNs, datasets with observations, and creating various kinds of static and dynamic properties. Once you have created the specification, you can start the inference window. There, you can then run the inference algorithms and explore or export 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.
- The overall workflow of the tool is shortly introduced in Section 2.
- We describe the Editor Session of the tool and its features in detail in Section 3.
- The Inference Session and results are described in detail in Section 4.
- A brief tutorial showcasing the main functionality on a simple model is presented in Section 5.
- Supported formats for datasets, sketches, and syntax for various kinds of supported expressions are discussed in Section 6.

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 the MIT license) is freely available at our GitHub: `https://github.com/sybila/biodivine-sketchbook`.

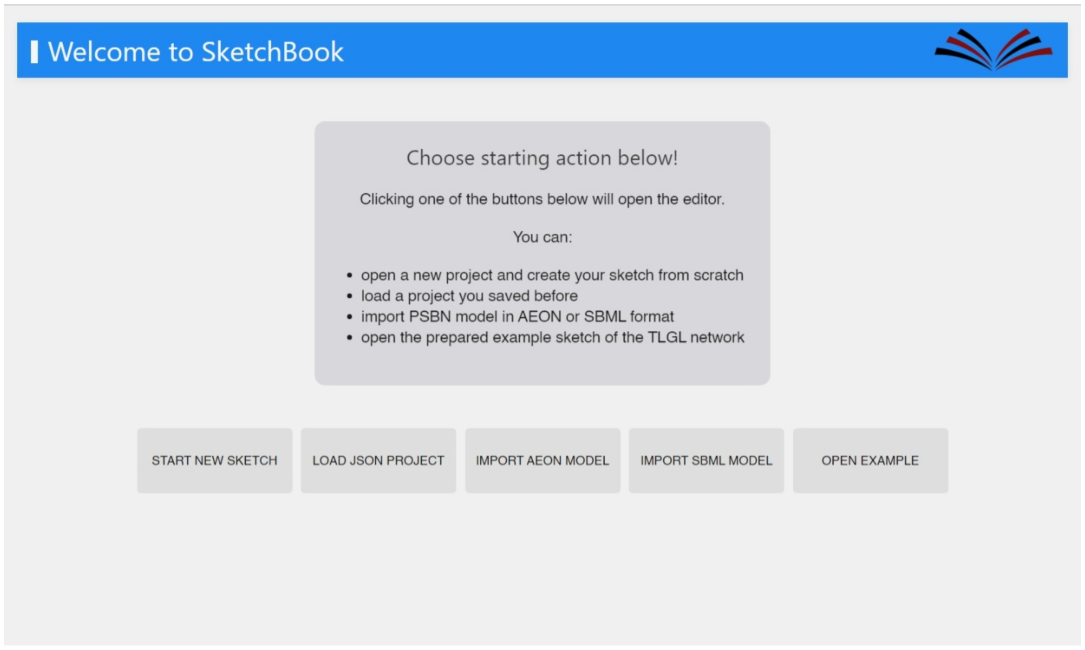
We provide pre-built binaries and installation files for the application in the release section of the repository. To start using Sketchbook, choose the latest release and download a binary for your operating system. You choose between `.app` and `.dmg` for **macOS**, `.AppImage`, `.deb` and `.rpm` for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

Note that on newer Linux distributions (Ubuntu 24 or Debian 13), some Tauri dependencies may not be supported by default. This does not affect the setup through the `.AppImage` (as it bundles all dependencies inside). However, it may affect installation with the `.deb` package, where you need to first install `libwebkit2gtk-4.0`. See a solution for that at this GitHub issue. This will be resolved once we switch to Tauri 2.

When installing the binaries, note that they 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:


Welcome to SketchBook

Choose starting action below!

Clicking one of the buttons below will open the editor.

You can:

- open a new project and create your sketch from scratch
- load a project you saved before
- import PSBN model in AEON or SBML format
- open the prepared example sketch of the TLGL network

Initial screen of Sketchbook.

2 Sketchbook Workflow

As discussed in the introduction, Sketchbook operates through two main sessions: the Editor Session, where you create and validate the model specification (i.e., design the Boolean network sketch), and the Inference Session, where you compute and explore all network candidates consistent with the specification. Each session operates independently, maintaining its own lifecycle and internal state. The main workflow proceeds as follows, with details on both sessions provided in the subsequent sections.

First, you design the influence network by defining variables and their regulatory interactions in the Network tab (see Section 3.2). Each regulation can be annotated with its monotonicity (activation, inhibition, dual, or unknown) and essentiality (essential, non-essential, or unknown).

Next, you can assign each variable a partially specified update function in the Functions tab (Section 3.3). These can be standard logical expressions, but may also involve supplementary uninterpreted functions representing unknown parts of the update mechanism to be determined by the inference. You can specify additional properties of these partially specified functions in the Properties tab, as discussed below.

You may import and edit binarised experimental datasets in the Observations tab (Section 3.4). The observations may come from steady-state or time-series expression measurements. These observations are later incorporated into dynamic properties (such as fixed points or trajectories) to constrain model behaviour in the Properties tab, as discussed below.

Then, you can define various required model properties in the Properties tab (Section 3.5). Static properties restrict the structure of the model (constraints on admissible update functions), while dynamic properties capture expected model behaviour. Dynamic properties can directly incorporate datasets, for example, to link observations to fixed points, attractors, trap spaces, or trajectories. You can either use predefined property templates or specify the properties directly as logical formulas.

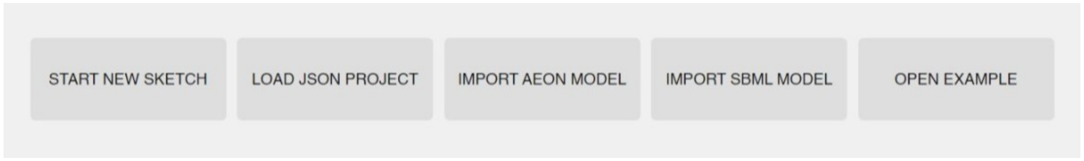
Before running inference, you can run a static validation check on the overall consistency of the sketch in the Analysis tab (Section 3.7). Detected issues can be reviewed and corrected directly in the editor.

Finally, once you finish editing the specification, start the Inference Session (Section 4) from the Analysis tab. A new window appears, where you can run the computation. The tool will compute all Boolean network models consistent with the specified sketch at once. Once inference finishes, you can review the summary of results (e.g., the summary of admissible functions), sample admissible networks, or export the compact symbolic encoding of the results for further analysis and refinement.

Note that once an Inference Session starts, any changes made in the editor afterwards will not affect that session. Each session works with a fixed snapshot of the model and its settings taken at the time it was created. You can, however, create multiple inference sessions simultaneously, each with its own inputs or configurations.

3 Sketch Editor Session

The sketch editor is the primary session of the tool that allows editing all components of the sketch. The tool starts with a simple initial screen where you can select the 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:

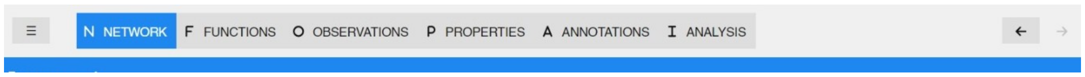


Available buttons at the initial screen.

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

3.1 General navigation

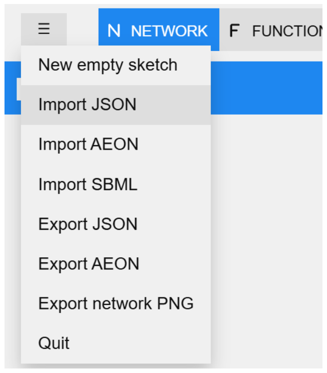
At the top of the screen, there is a pop-up menu, a list of tabs, and an undo-redo button.



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 support exporting the sketch in JSON and AEON. You can also export the network image as a PNG. More details on the formats are in Section 6. Note that importing a new sketch results in losing current data (you’ll get a warning).

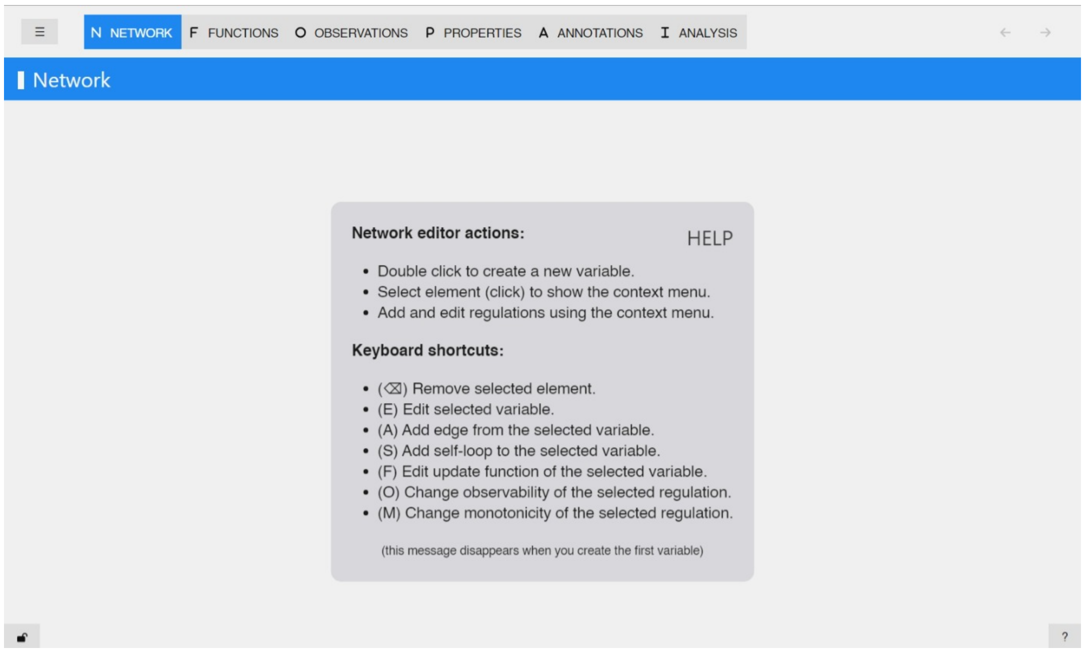


Menu with all the options.

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

3.2 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.



Network editor actions:

- Double click to create a new variable.
- Select element (click) to show the context menu.
- Add and edit regulations using the context menu.

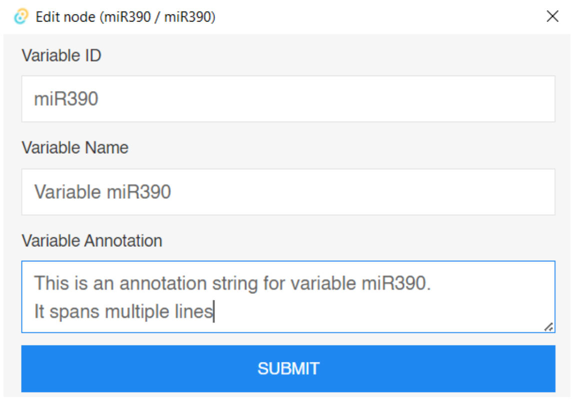
Keyboard shortcuts:

- (⌫) Remove selected element.
- (E) Edit selected variable.
- (A) Add edge from the selected variable.
- (S) Add self-loop to the selected variable.
- (F) Edit update function of the selected variable.
- (O) Change observability of the selected regulation.
- (M) Change monotonicity of the selected regulation.

(this message disappears when you create the first variable)

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

A new variable can be added by double-clicking in 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.



Variable ID: miR390

Variable Name: Variable miR390

Variable Annotation: This is an annotation string for variable miR390.
It spans multiple lines

Example of a dialogue window to edit a variable.

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

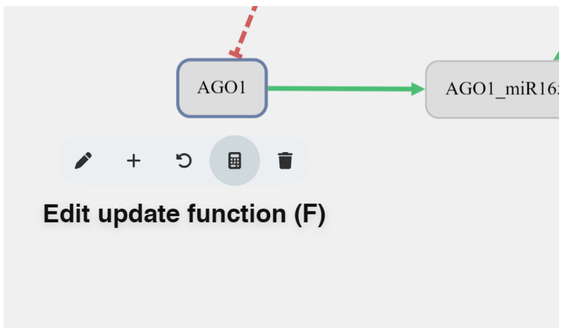
Monotonicity allows us to specify what type of effect the interaction has. The monotonicity of the regulation is reflected by its colour. The options are:

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

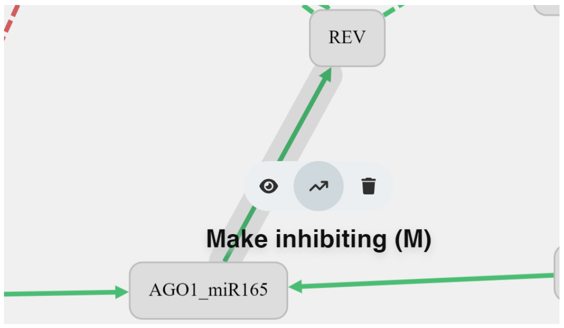
- **dual** for non-monotonous effect (blue colour)
- **unknown** for undefined (grey color)

Essentiality allows us 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)

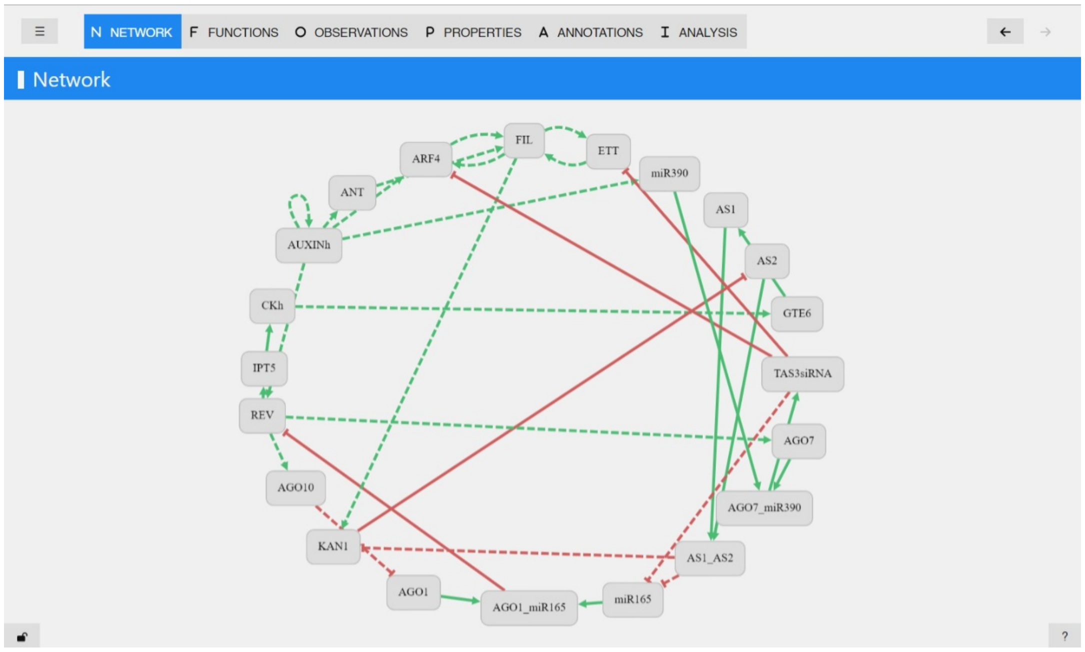


Variable context menu.



Regulation context menu.

The fully edited network can look like the following:

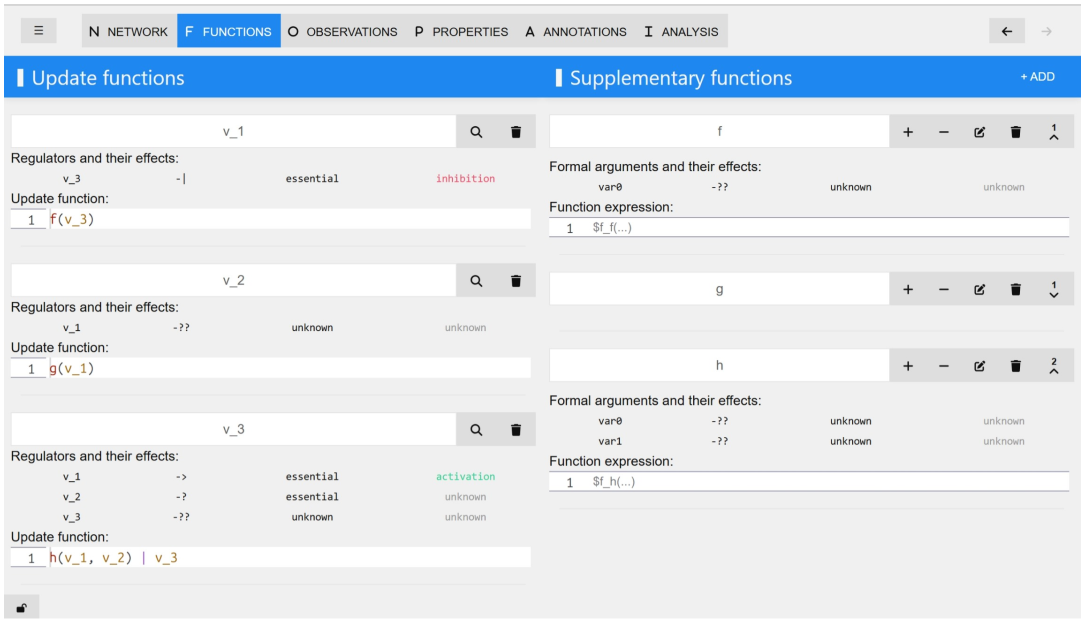


Network tab with an example of a created regulation network.

3.3 Editing update functions

Logical 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.



v_1

Regulators and their effects:

v_3	-|	essential	inhibition

Update function:

```
f(v_3)
```

v_2

Regulators and their effects:

v_1	-??	unknown	unknown

Update function:

```
g(v_1)
```

v_3

Regulators and their effects:

v_1	->	essential	activation
v_2	-?	essential	unknown
v_3	-??	unknown	unknown

Update function:

```
h(v_1, v_2) | v_3
```

f

Formal arguments and their effects:

var0	-??	unknown	unknown

Function expression:

```
$f_f(...)
```

g

h

Formal arguments and their effects:

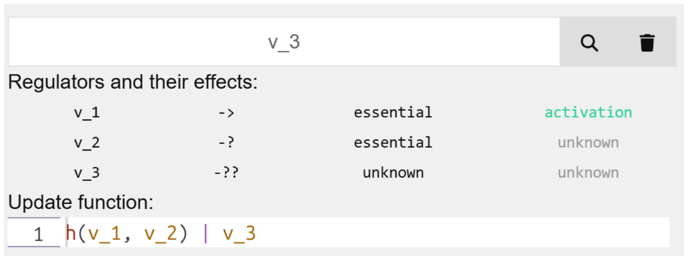
var0	-??	unknown	unknown
var1	-??	unknown	unknown

Function expression:

```
$f_h(...)
```

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 6.1.1.



v_3

Regulators and their effects:

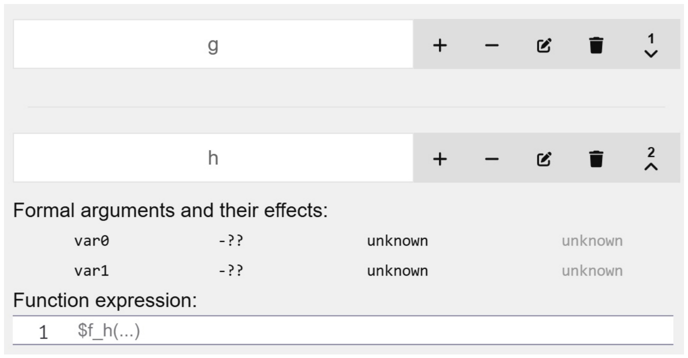
v_1	->	essential	activation
v_2	-?	essential	unknown
v_3	-??	unknown	unknown

Update function:

```
h(v_1, v_2) | v_3
```

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 or decrement its arity, update monotonicity/essentiality properties, set its partial expression, edit its details (same as for a variable, via an external dialogue), or delete it. The details of a supplementary function can be hidden by collapsing it.



Formal arguments and their effects:

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

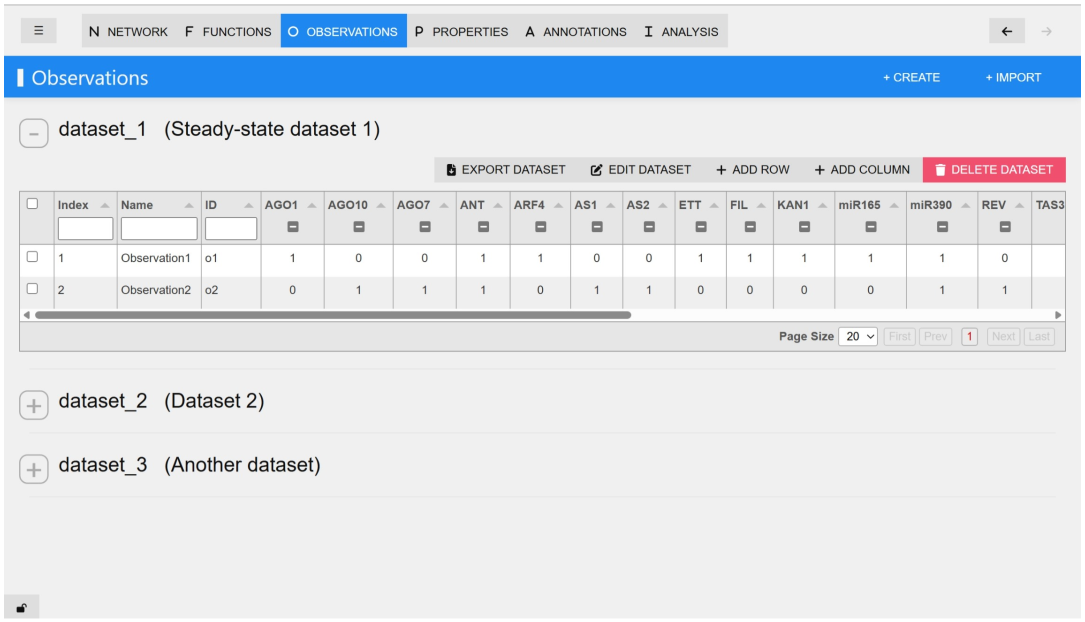
Function expression:

```
$f_h(...)
```

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

3.4 Loading and editing datasets

Binarised expression data can be imported and edited in the *Observations* tab. Each observation is an assignment of binarised values to the variables. Some values may be left unspecified. These datasets are later incorporated into dynamic properties (by linking observations to model fixed points, trajectories, and so on) in the Properties tab (see Section 3.5).



dataset_1 (Steady-state dataset 1)

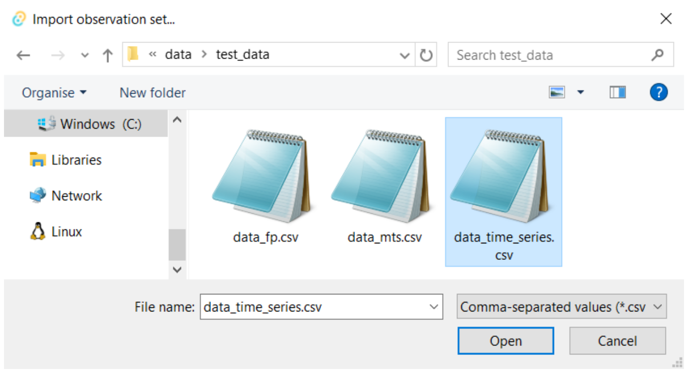
Index	Name	ID	AGO1	AGO10	AGO7	ANT	ARF4	AS1	AS2	ETT	FIL	KAN1	miR165	miR390	REV	TAS3
1	Observation1	o1	1	0	0	1	1	0	0	1	1	1	1	1	0	
2	Observation2	o2	0	1	1	1	0	1	1	0	0	0	0	1	1	

dataset_2 (Dataset 2)

dataset_3 (Another dataset)

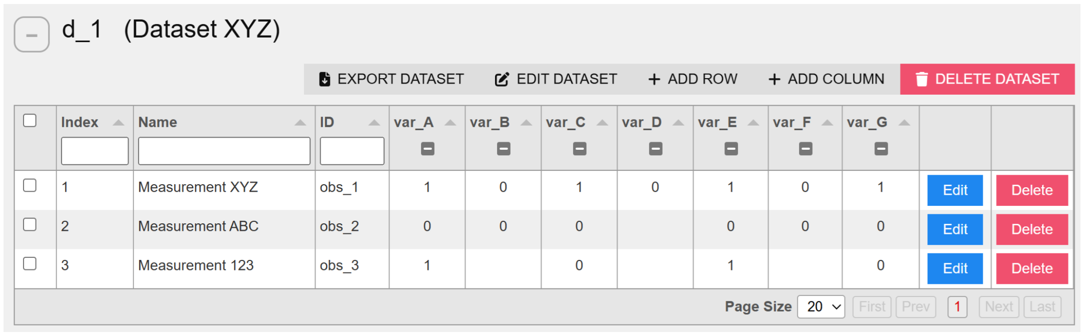
Observations tab with three datasets, one of them being expanded.

You can either import a `CSV` dataset (in the format discussed in Section 6) by clicking the `+IMPORT` button at the top or create one from scratch (clicking on the `+CREATE` button). The import button opens a file dialogue, and you can choose the CSV file 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. Datasets can be exported into CSV (to be reused in another sketch). 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 individual observations or whole datasets by clicking the corresponding pink `DELETE` buttons. You can also delete variables by left-clicking the column's heading and choosing the `Remove column` option.

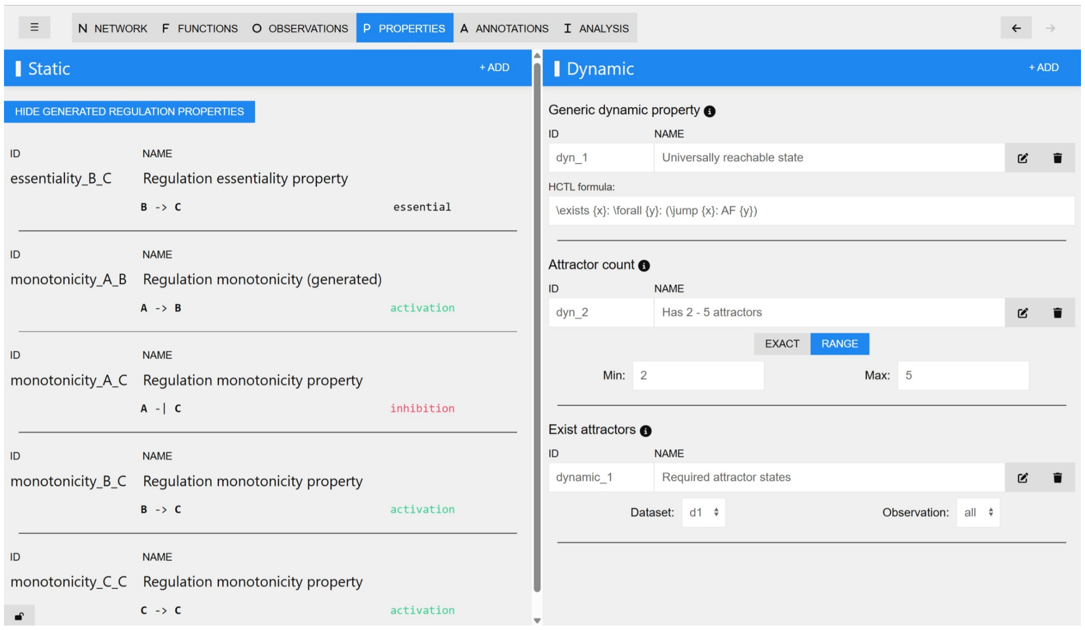


	Index	Name	ID	var_A	var_B	var_C	var_D	var_E	var_F	var_G		
☐	1	Measurement XYZ	obs_1	1	0	1	0	1	0	1	Edit	Delete
☐	2	Measurement ABC	obs_2	0	0	0		0	0	0	Edit	Delete
☐	3	Measurement 123	obs_3	1		0		1		0	Edit	Delete

Zoom to a table-based editor for a dataset.

3.5 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 the admissible update functions and regulations) while dynamic properties specify how the model must behave. Dynamic properties may incorporate previously defined datasets and observations.



Static + ADD

HIDE GENERATED REGULATION PROPERTIES

ID	NAME		
essentiality_B_C	Regulation essentiality property	`B -> C`	essential
monotonicity_A_B	Regulation monotonicity (generated)	`A -> B`	activation
monotonicity_A_C	Regulation monotonicity property	`A -| C`	inhibition
monotonicity_B_C	Regulation monotonicity property	`B -> C`	activation
monotonicity_C_C	Regulation monotonicity property	`C -> C`	activation

Dynamic + ADD

Generic dynamic property

ID	NAME
dyn_1	Universally reachable state

HCTL formula: \exists {x}: \forall {y}: (\jump {x}: AF {y})

Attractor count

ID	NAME
dyn_2	Has 2 - 5 attractors

EXACT RANGE

Min: 2 Max: 5

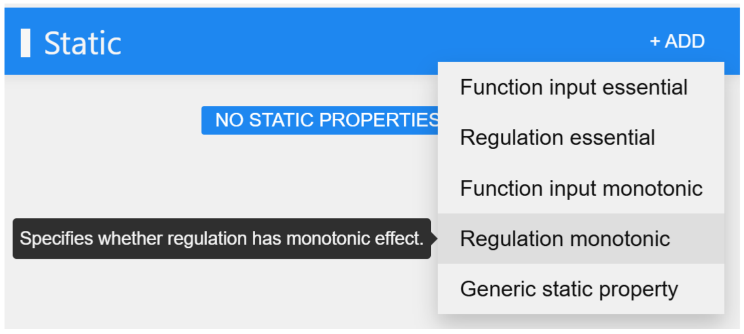
Exist attractors

ID	NAME
dynamic_1	Required attractor states

Dataset: d1 Observation: all

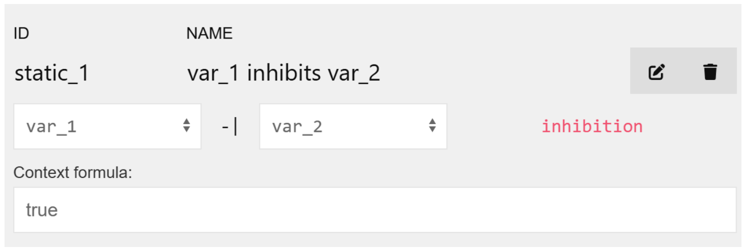
Properties tab with five automatically derived static properties and three dynamic properties.

For static properties, you can write a `generic` property in first-order logic (FOL); the syntax is described in Section 6.1.3). 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.



Specifies whether regulation has monotonic effect.

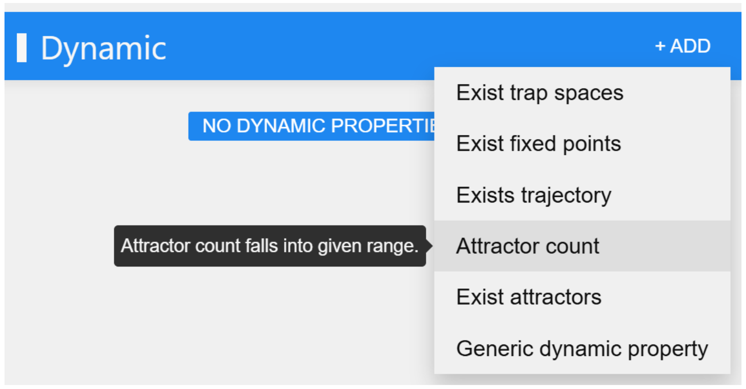
Template selection for static properties (with a tooltip help displayed for one of the templates).



Example of a regulation monotonicity property.

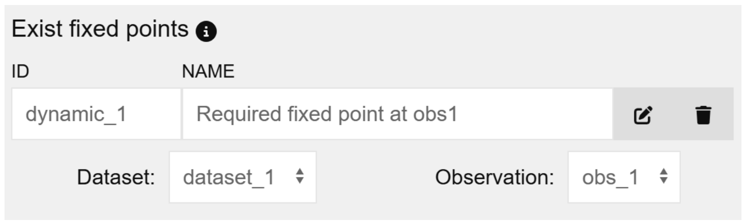
For dynamic properties, you can write a generic property in hybrid CTL (HCTL); the syntax is described in Section 6.1.2). We also offer a list of predefined templates for dynamic properties, e.g., to incorporate experimental data. There are templates for the following options (with more details following below):

- *Exist fixed points*: each selected observation corresponds to a fixed point
- *Exist attractors*: each selected observation corresponds to an attractor
- *Exist trap spaces*: each selected observation corresponds to a trap space
- *Exists trajectory*: observations of selected dataset lay on a trajectory
- *Attractor count*: attractor count falls into a given range



Template selection for dynamic properties (with a tooltip help displayed for one of the templates).

For the templates regarding the existence of *attractors* 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 selected observation that belongs to an attractor (or is a fixed point)." If the observation is fully specified (no missing values), it corresponds to exactly one state of the model. If there are missing values in an observation, it represents a whole subspace of the model, and we require that *some* state in that subspace must belong to an attractor (or be a fixed point). If no observation is selected in the template, we consider all observations in the dataset, and the individual formulas are combined into a conjunction.


Exist fixed points

ID	NAME
dynamic_1	Required fixed point at obs1

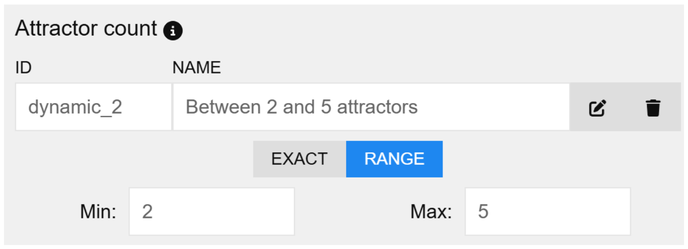
Dataset: dataset_1 Observation: obs_1

Example of a fixed-point property.

The template for existence of *trap spaces* also lets you select a dataset and its observation. The property then ensures that “There exists a trap space corresponding to the subspace given by the selected observation.”. Similarly to the previous, if the observation is omitted in the template, all observations in the selected datasets are considered. The trap space property template also lets you specify whether the trap spaces have to be *minimal* or *non-percolable* (not required by default).

The *trajectory* property can be used to encode time series measurements. It ensures the existence of a trajectory between states corresponding to the subsequent observations in a given dataset.

For the *attractor count* property, the user can specify how many attractors the model should admit in total. Users can select both exact number of attractor, or a range that the number must fall into.


Attractor count

ID	NAME
dynamic_2	Between 2 and 5 attractors

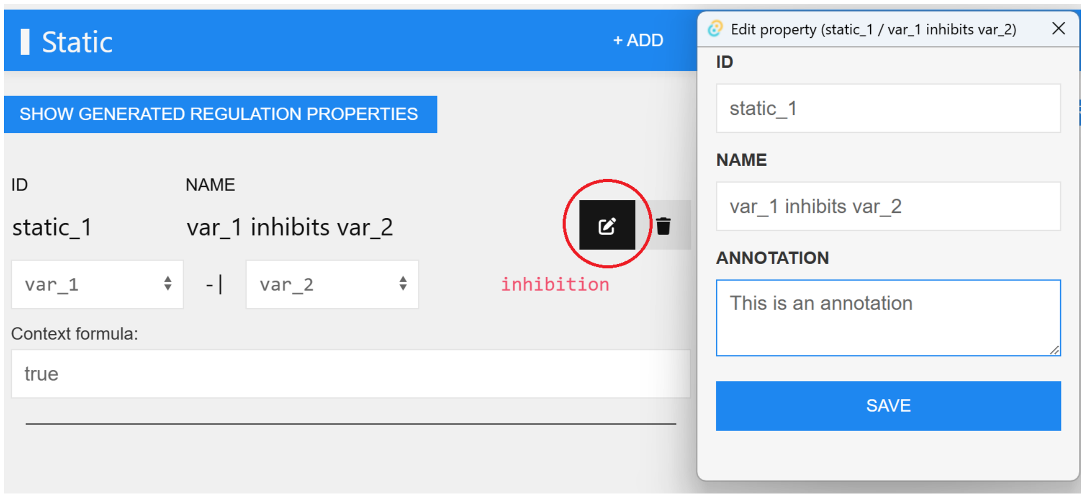
EXACT [✓ RANGE]

Min: 2 Max: 5

Example of an attractor count property.

3.6 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

SHOW GENERATED REGULATION PROPERTIES

ID	NAME
static_1	var_1 inhibits var_2

var_1 -| var_2 inhibition

Context formula:

true

Edit property (static_1 / var_1 inhibits var_2)

ID

static_1

NAME

var_1 inhibits var_2

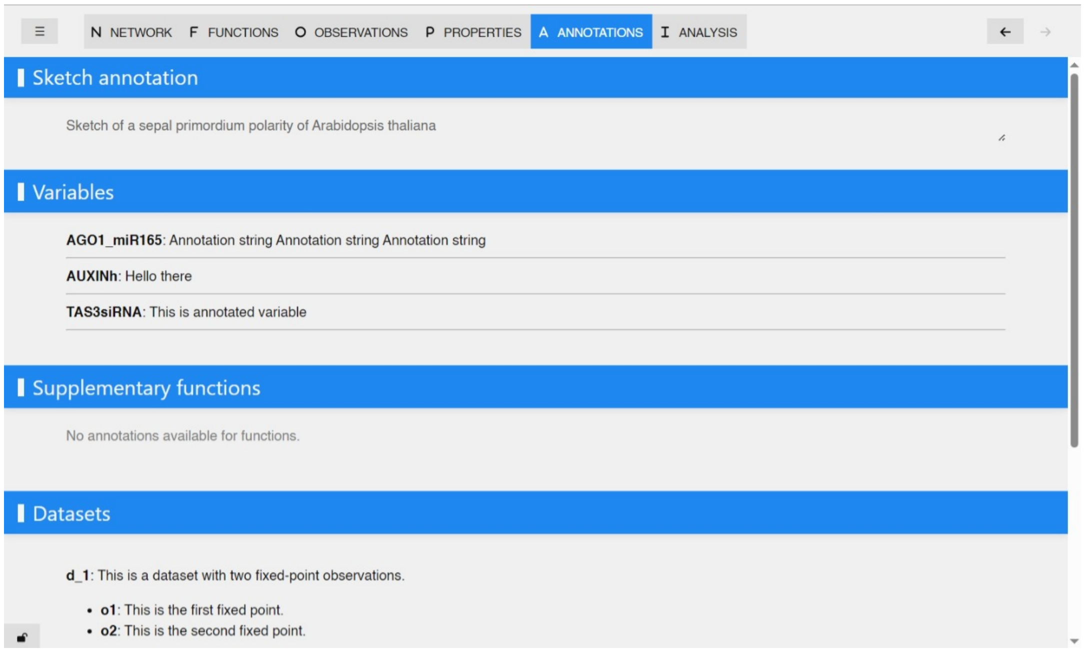
ANNOTATION

This is an 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.



Sketch annotation

Sketch of a sepal primordium polarity of Arabidopsis thaliana

Variables

AGO1_miR165: Annotation string Annotation string Annotation string

AUXINh: Hello there

TAS3siRNA: This is annotated variable

Supplementary functions

No annotations available for functions.

Datasets

d_1: This is a dataset with two fixed-point observations.

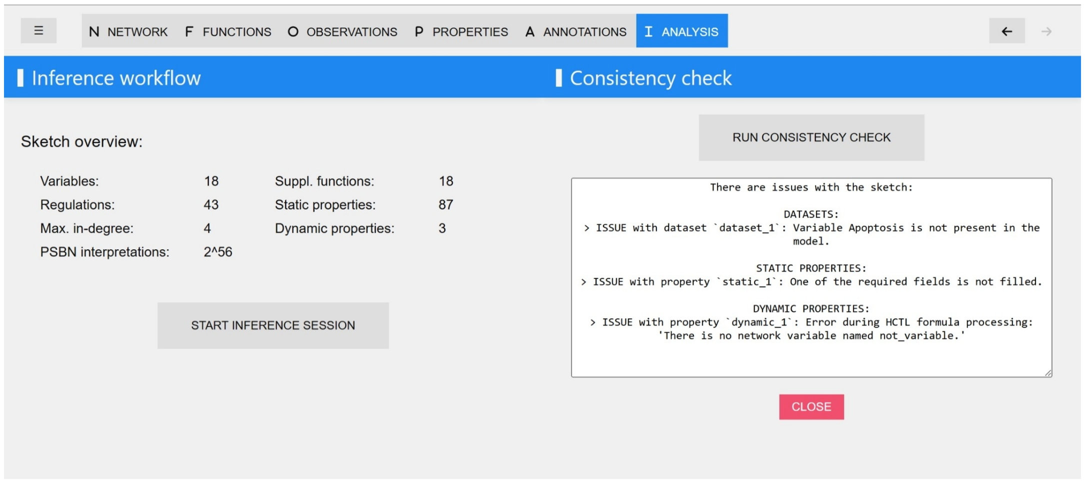
- **o1**: This is the first fixed point.
- **o2**: This is the second fixed point.

Annotations tab with annotations divided by entity types.

3.7 Starting analysis and checking consistency

The last part of the editor session is the *Analysis* tab. The current state is very simple. On the left, you can see a short summary of the sketch, and you can start the inference session. On the right, you can explicitly check the consistency of the sketch.

Most of the user inputs in all tabs are immediately automatically validated. If an issue is encountered with a user action, the corresponding action will not be executed, and the user will 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 consistency check button. If some minor non-critical issues are encountered (such as if a dataset is defined but not utilised in any dynamic properties), a simple warning message will pop up. Users can decide whether it is relevant and whether they should address it or not.



Sketch overview:

Variables:	18	Suppl. functions:	18
Regulations:	43	Static properties:	87
Max. in-degree:	4	Dynamic properties:	3
PSBN interpretations:	2^56		

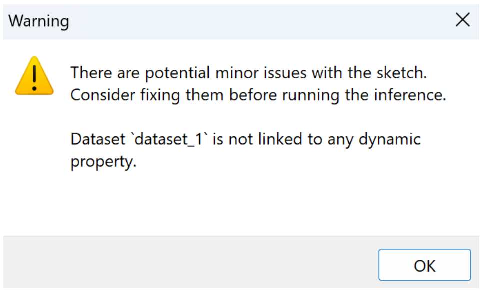
```
There are issues with the sketch:

DATASETS:
> ISSUE with dataset `dataset_1`: Variable Apoptosis is not present in the
model.

STATIC PROPERTIES:
> ISSUE with property `static_1`: One of the required fields is not filled.

DYNAMIC PROPERTIES:
> ISSUE with property `dynamic_1`: Error during HCTL formula processing:
'There is no network variable named not_variable.'
```

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



There are potential minor issues with the sketch.
Consider fixing them before running the inference.

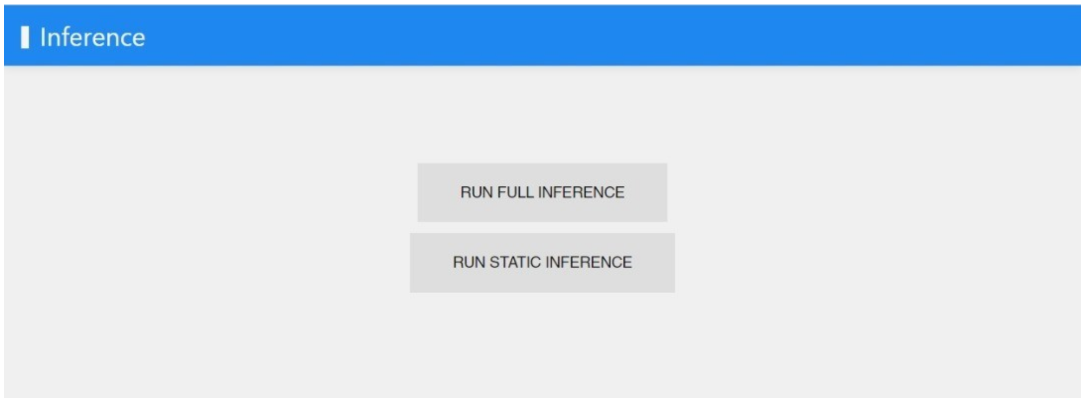
Dataset `dataset_1` is not linked to any dynamic property.

Example of a consistency warning.

Finally, after clicking on the `START INFERENCE SESSION`, a new window opens. We go into detail on the inference session functionality in Section 4. 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.

4 Inference Session

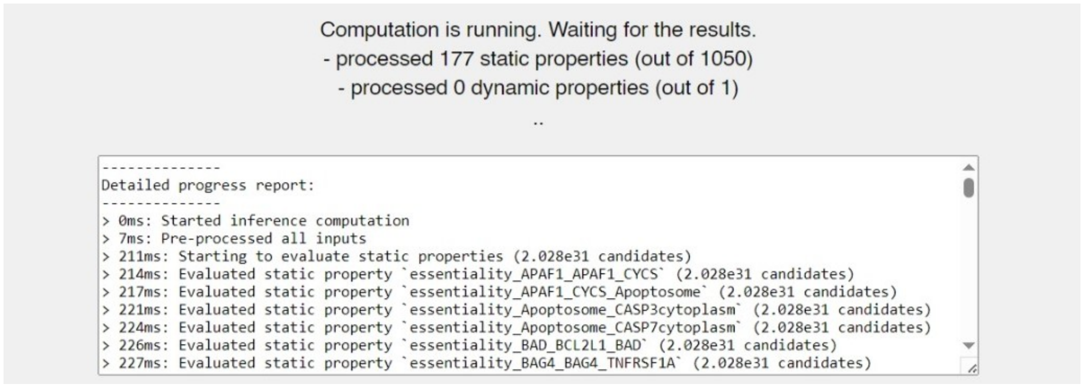
After starting the new inference session, a simple window appears where you can run the inference. You can choose from two options: running the full inference with all properties or performing a preliminary step – a partial “static” inference with static properties only. The static properties are usually 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.



Inital 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.



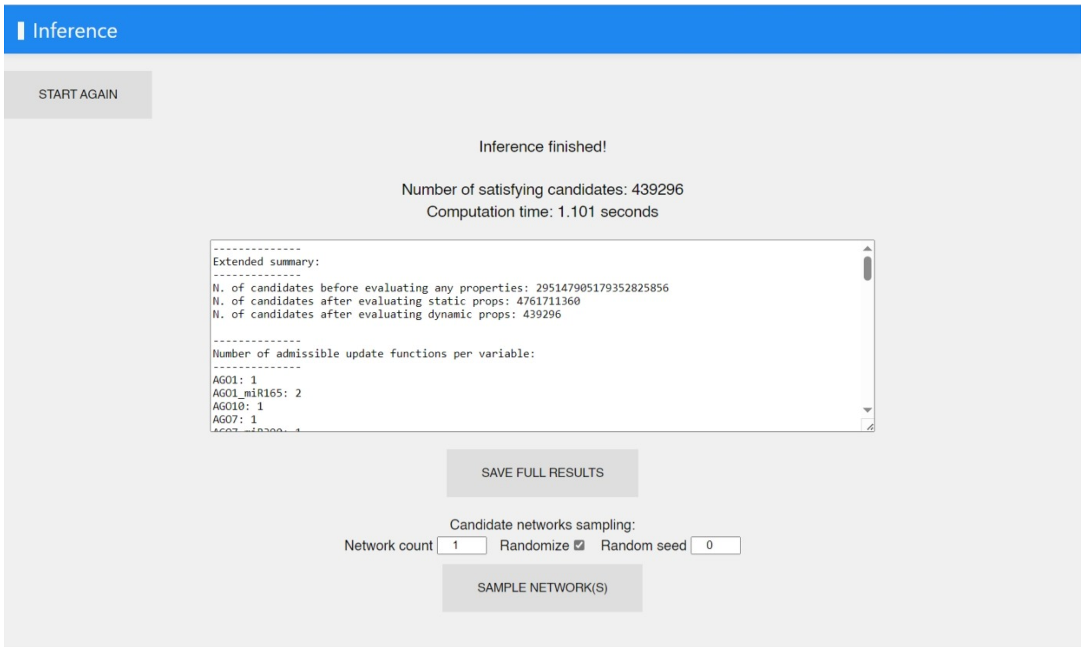
Computation is running. Waiting for the results.
- processed 177 static properties (out of 1050)
- processed 0 dynamic properties (out of 1)

..

```
--------------
Detailed progress report:
--------------
> 0ms: Started inference computation
> 7ms: Pre-processed all inputs
> 211ms: Starting to evaluate static properties (2.028e31 candidates)
> 214ms: Evaluated static property `essentiality_APAF1_APAF1_CYCS` (2.028e31 candidates)
> 217ms: Evaluated static property `essentiality_APAF1_CYCS_Apoptosome` (2.028e31 candidates)
> 221ms: Evaluated static property `essentiality_Apoptosome_CASP3cytoplasm` (2.028e31 candidates)
> 224ms: Evaluated static property `essentiality_Apoptosome_CASP7cytoplasm` (2.028e31 candidates)
> 226ms: Evaluated static property `essentiality_BAD_BCL2L1_BAD` (2.028e31 candidates)
> 227ms: Evaluated static property `essentiality_BAG4_BAG4_TNFRSF1A` (2.028e31 candidates)
```

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 finished!

Number of satisfying candidates: 439296
Computation time: 1.101 seconds

```
--------------
Extended summary:
--------------
N. of candidates before evaluating any properties: 295147905179352825856
N. of candidates after evaluating static props: 4761711360
N. of candidates after evaluating dynamic props: 439296

--------------
Number of admissible update functions per variable:
--------------
AGO1: 1
AGO1_miR165: 2
AGO10: 1
AGO7: 1
```

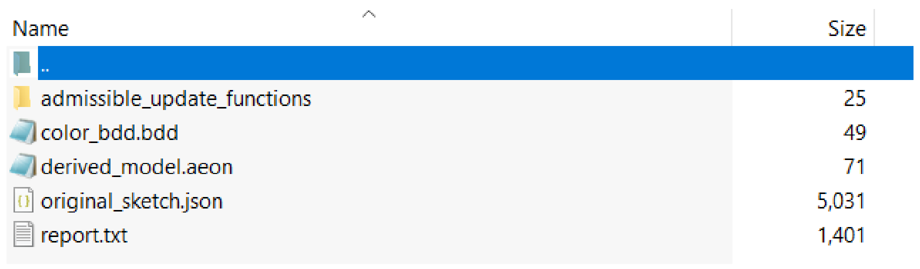
Candidate networks sampling:
Network count 1 Randomize ✓ Random seed 0

Inference window with a summary and buttons to ex- port/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:

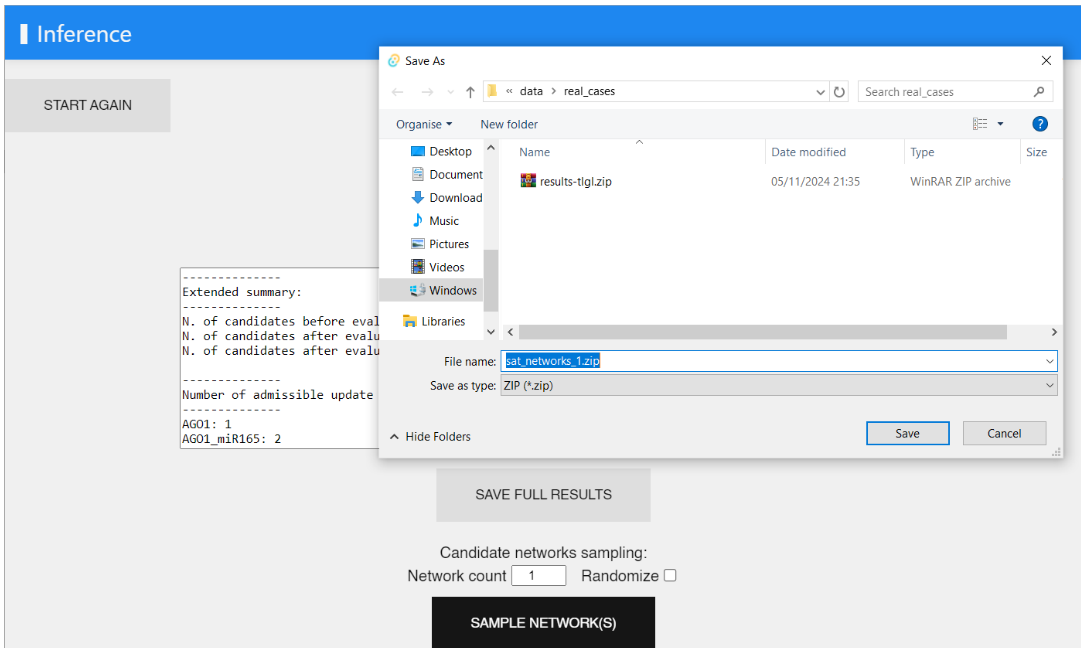
- A serialized BDD encoding all the admissible BN interpretations. The BDD format is compatible with the format used by the `aeon.py` library [1].
- Original sketch used as an input for the inference. This is included for replicability.
- Derived parametrized `aeon` model used internally during the computation. 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.



```
--------------
Extended summary:
--------------
N. of candidates before eval
N. of candidates after evalu
N. of candidates after evalu

--------------
Number of admissible update
--------------
AG01: 1
AG01_miR165: 2
```

Name	Date modified	Type	Size
results-tlgl.zip	05/11/2024 21:35	WinRAR ZIP archive	

File name: sat_networks_1.zip

Save as type: ZIP (*.zip)

Candidate networks sampling:

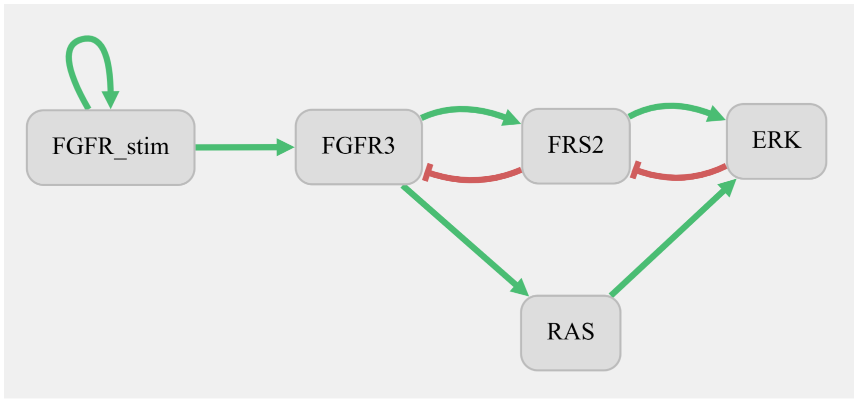
Network count 1 Randomize ☐

Example of a sampling dialogue.

5 Tutorial

To illustrate the main workflow and functionality of SKETCHBOOK, we use a toy version of the FGF signalling network, inspired by the model introduced in [4]. Note that the model is intentionally simplified and serves only to illustrate the main features of the tool. Most of the regulations are mediated by other entities or whole cascades in the real system. We also do not go into unnecessary details regarding the tool's interface (such as various buttons or options), as the functionality is introduced in previous sections.

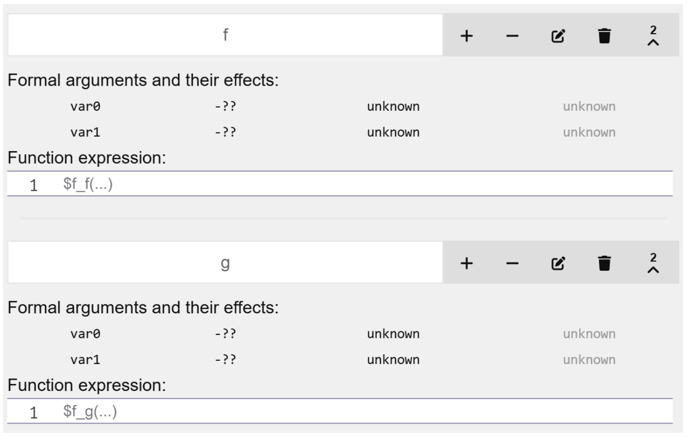
We start by creating the following influence graph in the Network Editor. It contains 5 variables: `FGFR_stim`, `FGFR3`, `FRS2`, `ERK`, and `RAS`. The `FGFR_stim` is modelled as an input node that activates the `FGFR3` receptor. The input has a single self-activating regulation to ensure it does not change value. This way, we also ensure that both values of the input are possible in all model interpretations. The `FGFR3` can then activate `ERK` either through `FRS2` or `RAS` (both representing whole activation cascades). There is a negative feedback from `FRS2` back to `FGFR3` (mediated by `GRB2` in the original model) and from `ERK` back to `FRS2` (mediated by `SPRY` in the original model).



Next, we assign the partial update functions (considering prior knowledge). For the variables `FGFR_stim` and `RAS`, there is only one way to set up the functions, so we fill them directly. Next, suppose that `FGFR3` requires both the presence of `FGFR_stim` and the absence of `FRS2`. Lastly, let's assume the particular update rules of `ERK` and `FRS2` are unknown and to be determined by the inference. We can assign them binary function symbols for now, and the inference process will find satisfying interpretations of these functions. Together, we get the following PSBN:

$$
\begin{aligned}
F_{\texttt{FGFR\_stim}} &= \texttt{FGFR\_stim}\\
F_{\texttt{FGFR3}} &= \texttt{FGFR\_stim} \land \neg \texttt{FRS2}\\
F_{\texttt{RAS}} &= \texttt{FGFR3}\\
F_{\texttt{ERK}} &= \mathbf{f}(\texttt{RAS}, \texttt{FRS2})\\
F_{\texttt{FRS2}} &= \mathbf{g}(\texttt{FGFR3}, \texttt{ERK})
\end{aligned}
$$

We can specify this PSBN in the Function Editor. First, we have to add the two supplementary functions **f** and **g**, both with arity 2 (and we do not select any of their properties).



f

Formal arguments and their effects:

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

Function expression:

```
$f_f(...)
```

g

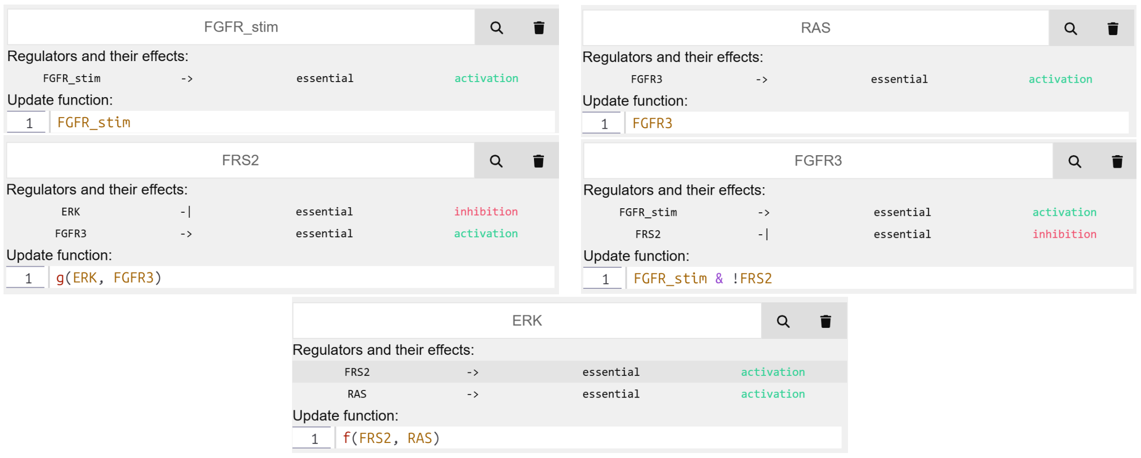
Formal arguments and their effects:

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

Function expression:

```
$f_g(...)
```

We can then set all the update functions as described above:



FGFR_stim

Regulators and their effects:

FGFR_stim	->	essential	activation

Update function:

```
FGFR_stim
```

RAS

Regulators and their effects:

FGFR3	->	essential	activation

Update function:

```
FGFR3
```

FRS2

Regulators and their effects:

ERK	-|	essential	inhibition
FGFR3	->	essential	activation

Update function:

```
g(ERK, FGFR3)
```

FGFR3

Regulators and their effects:

FGFR_stim	->	essential	activation
FRS2	-|	essential	inhibition

Update function:

```
FGFR_stim & !FRS2
```

ERK

Regulators and their effects:

FRS2	->	essential	activation
RAS	->	essential	activation

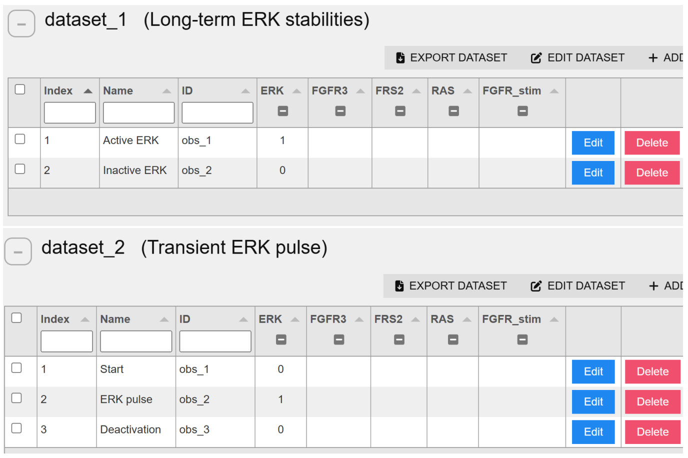
Update function:

```
f(FRS2, RAS)
```

Next, let's prepare some dynamic properties utilising simple observations. We will only consider measurements of `ERK` for this simple case (values of other variables can be left unspecified). First, it was measured that in the long term, `ERK` can stabilise in both active and inactive states. This gives us two observations, and we will use them to define model fixed points in the next step.

Furthermore, suppose we did simple time series measurements with `ERK` and observed that starting with the inactive form, it becomes active and then goes back to inactive. This gives us three *transient* observations that we will use to define a trajectory property of the model in the next step.

We can specify the measurements in the Observations Editor. Specifically, we create two datasets, one with steady state observations and one for the transient time series.



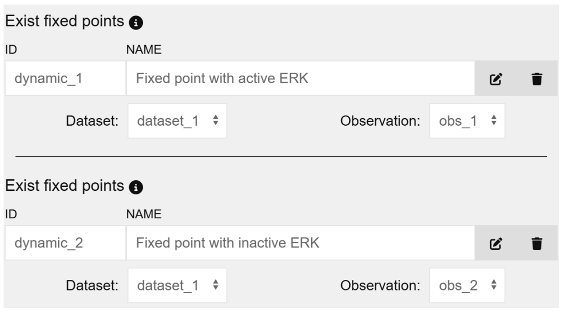
dataset_1 (Long-term ERK stabilities)

Index	Name	ID	ERK	FGFR3	FRS2	RAS	FGFR_stim
1	Active ERK	obs_1	1				
2	Inactive ERK	obs_2	0				

dataset_2 (Transient ERK pulse)

Index	Name	ID	ERK	FGFR3	FRS2	RAS	FGFR_stim
1	Start	obs_1	0				
2	ERK pulse	obs_2	1				
3	Deactivation	obs_3	0				

Then, in the Properties Editor, we link the observations to the corresponding required *dynamic properties*. We create two *Fixed points* properties for the two steady-state observations, and a *Trajectory* property to capture the transient ERK pulse.



Exist fixed points

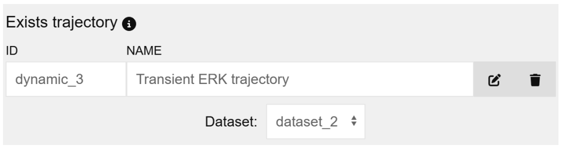
ID	NAME
dynamic_1	Fixed point with active ERK

Dataset: dataset_1 Observation: obs_1

Exist fixed points

ID	NAME
dynamic_2	Fixed point with inactive ERK

Dataset: dataset_1 Observation: obs_2

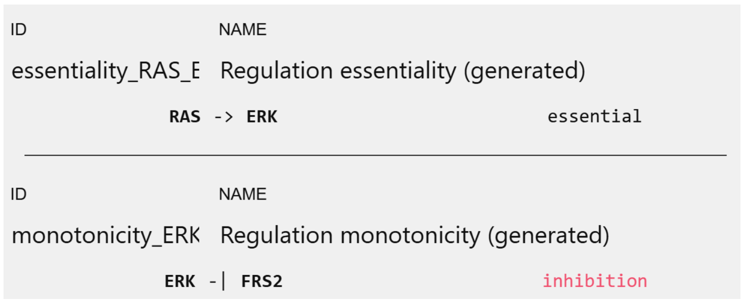


Exists trajectory

ID	NAME
dynamic_3	Transient ERK trajectory

Dataset: dataset_2

Furthermore, SKETCHBOOK automatically creates *static properties* encoding monotonicity and essentiality of the regulations (reflecting the Network Editor). Some of these are illustrated below. In this example, we do not add any additional static properties.



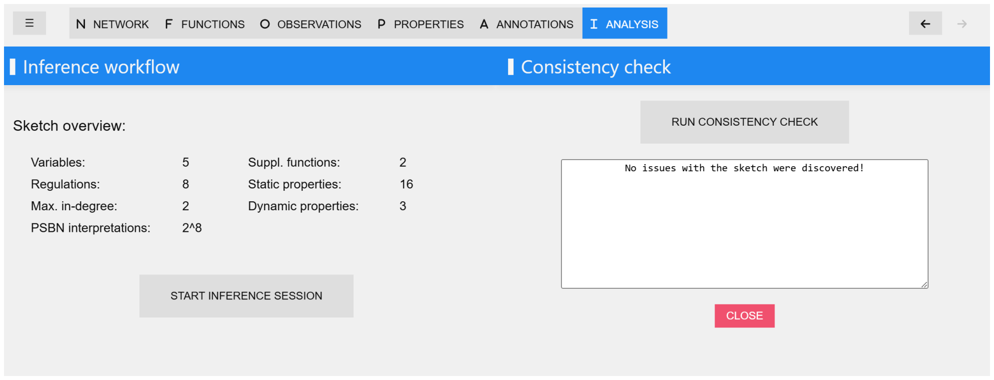
ID	NAME
essentiality_RAS_E	Regulation essentiality (generated)

RAS -> ERK essential

ID	NAME
monotonicity_ERK	Regulation monotonicity (generated)

ERK -| FRS2 inhibition

With that, we are ready to start the inference. In the Analysis Tab, we can see the overview of the sketch and run the consistency check. This check reports on potential syntactic issues (such as inconsistent variable names between the network and datasets). Once we have confirmed the consistency, we can start the inference session.

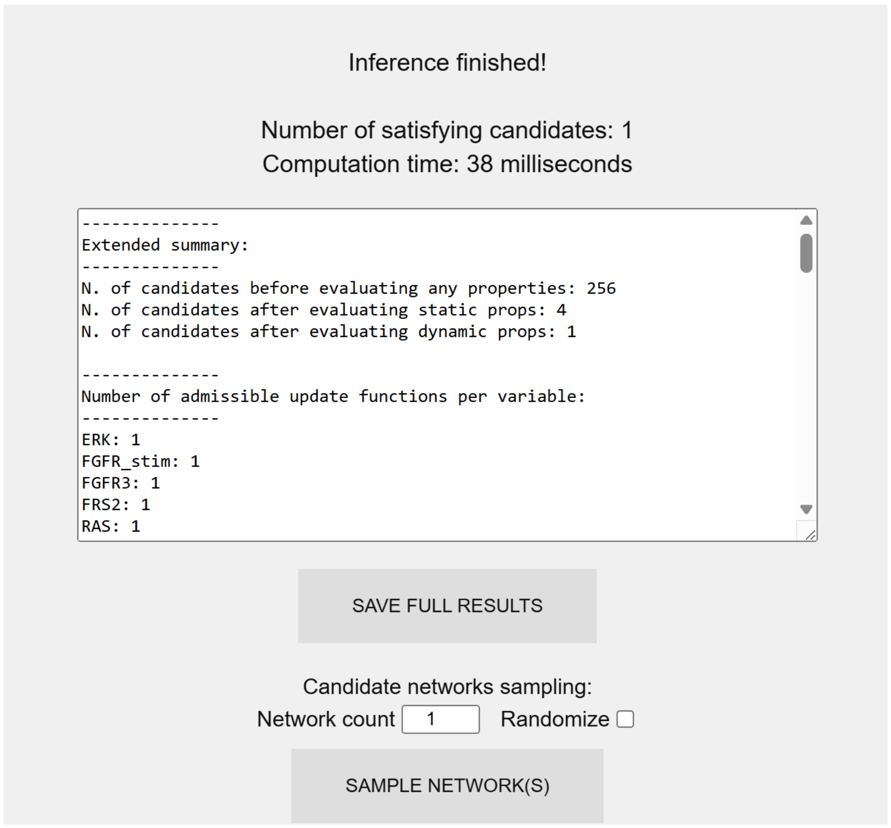


Sketch overview:

Variables:	5	Suppl. functions:	2
Regulations:	8	Static properties:	16
Max. in-degree:	2	Dynamic properties:	3
PSBN interpretations:	2^8		

```
No issues with the sketch were discovered!
```

After starting the inference, a new window opens, taking the current state of the sketch from the editor as input. Once we start the inference, SKETCHBOOK computes all admissible Boolean networks consistent with the specification. The tool then displays the following summary of the computation and the resulting set of candidate BNs.



Inference finished!

Number of satisfying candidates: 1
Computation time: 38 milliseconds

```
--------------
Extended summary:
--------------
N. of candidates before evaluating any properties: 256
N. of candidates after evaluating static props: 4
N. of candidates after evaluating dynamic props: 1

--------------
Number of admissible update functions per variable:
--------------
ERK: 1
FGFR_stim: 1
FGFR3: 1
FRS2: 1
RAS: 1
```

Network count: 1

In this case, there is a single candidate consistent with the whole specification.

In the extended summary, we see how the number of solutions evolved at various stages of the computation. SKETCHBOOK starts the computation by symbolically encoding all interpretations of the input PSBN (before considering any static or dynamic properties). With two function symbols of arity two, we have $16 \times 16 = 256$ possible BNs. The tool then refines the set, filtering interpretations consistent with static properties (monotonicities and essentialities of the regulations).

After this, there are 4 BNs remaining (as expected, looking at the simple network). Finally, the set is further refined to only the subset of interpretations that are also consistent with all dynamic properties, leaving us with a single candidate model. We can also see the summary of admissible update functions across all the candidates. Since we only have a single candidate, only one version of each update function is possible.

We can either export the symbolic representation of the resulting set of all admissible models or sample the BNs. Since there is only one candidate, let's use the BN sampling. After downloading and opening the model file, we get the following BN in AEON format:

```
FRS2 -> ERK
RAS -> ERK
FGFR_stim -> FGFR3
FRS2 -| FGFR3
FGFR_stim -> FGFR_stim
ERK -| FRS2
FGFR3 -> FRS2
FGFR3 -> RAS
$ERK: RAS | FRS2
$FGFR3: FGFR_stim & !FRS2
$FGFR_stim: FGFR_stim
$FRS2: !ERK & FGFR3
$RAS: FGFR3
```

The first part summarises the regulations (same as in the specification) and the second part the update functions. We see that the inferred update function for `FRS2` is a conjunction $\neg$`ERK` $\vee$ `FGFR3`, and for `ERK` it is a disjunction `RAS` $\vee$ `FRS2`. The functions respect the selected regulation properties.

If you explore the model's state space, there are two fixed-point attractors, one with active `ERK` and the other with inactive `ERK`, as we specified. Similarly, the model can also replicate the transient `ERK` pulse.

6 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. An annotation is a completely arbitrary string.

6.1 Logical expressions and languages

6.1.1 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.

$$\begin{aligned}
\mathit{function} &= \texttt{(}\ \mathit{function}\ \texttt{)} \mid \mathit{const} \mid \mathit{varId} \mid \mathit{fnSymbol} \mid \texttt{!}\mathit{function} \mid \mathit{function}\ \mathit{OP}\ \mathit{function}\\
\mathit{const} &= \texttt{true} \mid \texttt{false}\\
\mathit{OP} &= \texttt{\&} \mid \texttt{|} \mid \texttt{=>} \mid \texttt{<=>}\\
\mathit{fnSymbol} &= \mathit{fnId}\texttt{(}\mathit{args}\texttt{)}\\
\mathit{args} &= \mathit{function} \mid \mathit{args},\ \mathit{args}
\end{aligned}$$

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

```
C => (B & h(A,C))
```

6.1.2 Syntax of generic dynamic properties

Users can define generic dynamic properties using the logic HCTL. For user convenience, we also support certain "template meta-expressions" (or shortcuts) corresponding to various property templates. This way, users can use shortcuts for trajectory or attractor formulas. We provide details on these shortcuts at the end of this subsection.

The following grammar defines our textual format for HCTL formulae:

$$\begin{aligned}
\mathit{formula} &= \texttt{(}\ \mathit{formula}\ \texttt{)} \mid \mathit{const} \mid \mathit{propName} \mid \mathit{var} \mid \mathit{unaryOp}\ \mathit{formula}\\
&\quad \mid \mathit{formula}\ \mathit{binaryOp}\ \mathit{formula}\\
&\quad \mid \mathit{formula}\ \mathit{binaryOp}\ \mathit{formula}\\
&\quad \mid \texttt{\%}\mathit{templateExpression}\texttt{\%}\\
\mathit{const} &= \texttt{true} \mid \texttt{false}\\
\mathit{var} &= \texttt{\{}\mathit{varName}\texttt{\}}\\
\mathit{unaryOp} &= \sim \mid \texttt{AX} \mid \texttt{EX} \mid \texttt{AF} \mid \texttt{EF} \mid \texttt{AG} \mid \texttt{EG}\\
\mathit{binaryOp} &= \texttt{\&} \mid \texttt{|} \mid \texttt{=>} \mid \texttt{<=>} \mid \texttt{\^{}} \mid \texttt{AU} \mid \texttt{EU}\\
\mathit{hybridOp} &= \texttt{!} \mid \texttt{3} \mid \texttt{V} \mid \texttt{@} \mid \texttt{\textbackslash bind} \mid \texttt{\textbackslash exists} \mid \texttt{\textbackslash forall} \mid \texttt{\textbackslash jump}
\end{aligned}$$

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 `^` 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, `V` for the universal quantifier, and `@` for the jump operator. Correspondence with other logical and temporal operators is straightforward. Lastly, *templateExpression* corresponds to previously mentioned template shortcut expressions listed at the end of this subsection.

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

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

Template shortcut expressions

Users can also use the following shortcut expressions corresponding to the property templates discussed in Section 3.5. These expressions must be enclosed in percentage characters in a formula, as shown in the grammar above. Similar to property templates provided by the GUI, these expressions are often parametrized, referencing dataset IDs or other arguments. We currently support the following template expressions:

- observation in a dataset given as `datasetId, observationId`
- trajectory template given as `trajectory(datasetId)`
- attractors template given as `attractors(datasetId, observationId)` or `attractors(datasetId)`
- fixed points template given as `fixed_points(datasetId, observationId)` or `fixed_points(datasetId)`
- general trap spaces template given as `trap_spaces(datasetId, observationId)` or `trap_spaces(datasetId)`
- minimal trap spaces template given as `min_trap_spaces(datasetId, observationId)` or `min_trap_spaces(datasetId)`
- non-percolable trap spaces template given as `non_percolable_trap_spaces(datasetId, observationId)` or `non_percolable_trap_spaces(datasetId)`
- attractor count given as `attractor_count(minimal, maximal)` or `attractor_count(number)`

Below is an example of a property utilizing these template shortcuts. In this scenario, the sketch must have a dataset `d` with observations `o1` and `o2`. The property expresses that either `o1` is in the model's only attractor or `o2` is a fixed point.

```
(%attractors(d, o1)% & %attractor_count(1)%) | (%fixed_points(d, o2)%)
```

6.1.3 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.

$$\begin{aligned}
\mathit{formula} &= \texttt{(}\ \mathit{formula}\ \texttt{)} \mid \mathit{const} \mid \mathit{var} \mid \mathit{fnSymbol} \mid \texttt{!}\mathit{formula} \\
&\quad \mid \mathit{formula}\ \mathit{binaryOp}\ \mathit{formula} \\
&\quad \mid \mathit{quantifier}\ \mathit{varList}\ \texttt{:}\ \mathit{formula} \\
\mathit{const} &= \texttt{true} \mid \texttt{false} \\
\mathit{binaryOp} &= \texttt{\&} \mid \texttt{|} \mid \texttt{=>} \mid \texttt{<=>} \mid \texttt{\^{}} \\
\mathit{fnSymbol} &= \mathit{fnId}\texttt{(}\mathit{args}\texttt{)} \\
\mathit{args} &= \mathit{function} \mid \mathit{args},\ \mathit{args} \\
\mathit{quantifier} &= \texttt{3} \mid \texttt{V} \mid \texttt{\textbackslash exists} \mid \texttt{\textbackslash forall} \\
\mathit{varList} &= \mathit{var} \mid \mathit{var}, \mathit{varList}
\end{aligned}$$

Note that *var* stands for a FOL variable, and *fnId* is the 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)
```

6.2 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 at 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.

6.3 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).

6.3.1 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.

6.3.2 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 6.1.1.

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).

6.3.3 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., Kadlecaj, 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)

[4] Grieco, L., Calzone, L., Bernard-Pierrot, I., Radvanyi, F., Kahn-Perlès, B., Thieffry, D.: Integrative modelling of the influence of mapk network on cancer cell fate decision. PLOS Computational Biology **9**(10) (2013)