# RE:IN Tutorial

The Reasoning Engine for Interaction Networks (RE:IN) is a tool that runs online in your web browser, which allows the user to encode an Abstract Boolean Network (ABN) together with a set of experimental constraints, and to synthesise and interrogate concrete Boolean networks (BNs) that satisfy the expected behaviour.

The methodology behind RE:IN is explained in our publication, “A Method to Identify and Analyze Biological Programs through Automated Reasoning” (under review) (*1*). The purpose of this tutorial is to guide users to implement their own problems within the tool.

For existing users of the tool, please note that as of February 2016, we have updated RE:IN. In addition to extended functionality, the updated version implements some minor syntax changes for interaction and constraint files. These changes are highlighted throughout the tutorial.

# Contents

## [Encoding a Set of Components and Interactions](#interactions)

## [Encoding a Set of Experimental Observations](#observations)

## [Finding Models that Satisfy Experimental Observations](#FindingModels)

## [When the Solver Cannot Find Any Models](#NoModels)

## [Required and Disallowed Interactions](#RequiredInteractions)

## [Minimal Models](#MinimalModels)

## [Experiments with Genetic Perturbations](#GeneticPerturbtions)

## [Making New Predictions](#NewPredictions)

## [Example Files](#SampleFiles)

## [References](#References)

# Encoding a Set of Components and Interactions

The BN model framework is general enough that the components of the network can correspond to genes, proteins or other interacting molecules. Here we outline the use of the tool for the case of a small set of interacting genes, A, B, C and D.

The set of interactions can be entered manually into RE:IN, or loaded from a text file. Both methods of input are described here, with manual entry first.

### Manually defining components



First you define the components in the network, followed by the set of interactions between them. Under the ‘Interactions’ tab on the left hand side, click the ‘Add Component’ button. To name this component, double click the ‘new node’ text and enter the text required, e.g. ‘A’. This process can be repeated for as many genes as necessary.

The ‘KO’ and ‘FE’ checkboxes correspond to whether this gene can be knocked out or overexpressed in any of the experimental observations. Please refer to the section on [genetic perturbations](#GeneticPerturbtions) for when it is necessary to select these choices.

The ‘Needs Activator’ checkbox puts a restriction on concrete topologies that are derived – if checked, this specific component will have to have at least one activator (note that this does not restrict the state of the activator in a given experiment, nor the regulation conditions).

### Selecting regulation conditions

When you create a component, the ‘All Regulation Conditions’ checkbox is automatically selected. This refers to the set of Boolean update functions from which the solver can select, which define how the gene will update its state dependent on the presence or absence of its activators and inhibitors. For example, if A has two activators and one repressor, it may require both activators in order to be expressed, but B, which also has two activators and repressor, may need only one activator to be expressed. We define a set of update functions, summarised in Fig. 1 and referred to as regulation conditions, which reason over whether a gene has some, all or none of its activators or repressors. Regulation conditions 0 to 8 share the common assumption that a gene cannot be activated unless it has at least one activator that is expressed. These regulation conditions are described in detail in the RE:IN methodology paper (*1*).

In addition to the 16 we define, we also have implemented the threshold update function (*2*), in which the balance of activators and repressors determines whether the component is activated or repressed. The standard threshold rule is regulation condition 18, and the delayed threshold rule is regulation

condition 19. The second of these specifies that if a target component is active at time t, and its activators and repressors are balanced, then it will be degraded at time t+1.

Figure 1: **The set of update functions that RE:IN selects from for each component within the network.** The colour of the rectangles indicates the state of the component being regulated under the scenario indicated at the top of each column. Red indicates the component will be active, white indicates the component will be inactive. A subset of these functions can be assigned, should biological knowledge allow the user to restrict the set. For example, regulation conditions 0 to 8 ensure that a component cannot be activated without at least one of its activators being expressed. The yellow rectangles highlight that these conditions assume that in the presence of all activators and no repressors, a component will always be active, and conversely in the presence of all repressors and no activators, a component will always be inactive.

In this updated version of RE:IN, we have edited the regulation conditions for clarity. This updated set is the default set that is used. If you would prefer to use the regulation conditions defined in the first version of RE:IN (as implemented in our pluripotency program work (*3*)) then you can use the directive ‘Legacy’. If you wish to use only the regulation conditions in Fig. 1, use ‘NoThreshold’. These can be accessed via the ‘Options’ tab, where there is a drop down menu.

If you wish to restrict the set of regulation conditions further for a given component, uncheck the ‘All Regulation Conditions’ box and enter the appropriate numbers, as shown in the tool snapshot.

### Manually defining interactions

After defining the set of components, you can add interactions between them. For each interaction that is required, click the ‘Add Interaction’ button. A default interaction will appear in the bottom panel, which uses the first component name on your list. Use the drop-down menus to set the source and target of this interaction from the components you have already defined. If this interaction is positive, select the ‘Positive’ checkbox. Leave unchecked to define a negative interaction. Lastly, if this interaction is definite, select the ‘Definite’ checkbox. Leaving this box unchecked sets the interaction to be optional. If you have a model with a single interaction that is optional, you have automatically defined two possible topologies for the network – one in which the interaction is present, and one in which the interaction is absent.

### Deleting components and interactions

Both components and interactions can be deleted using the ‘X’ button on the appropriate line. If you delete a component, you automatically delete any interactions that require that component.

### Saving the set of interactions

Once the complete set of components and interactions have been defined, they can be saved to a text file. Click the ‘Save’ button, and enter the desired file name in the dialog box that pops up. The file extension for these files should be ‘.net’, but the solver will also accept ‘.txt’ files.

### Loading a prepared file of components, interactions and constraints

Saved files (.net files) can be opened by simply clicking the ‘Load’ button, and selecting the desired file. The component and interaction boxes will be populated automatically, and can be edited manually as above.

To prepare a model file in a text editor outside RE:IN, it must adopt the format shown in the Table 1. The text enclosed within the right-hand box below can be copied and saved, and then loaded into RE:IN. This is made available on the RE:IN website as the ‘toy’ example.

|  |  |
| --- | --- |
| First define the settings: i) whether you want to use the ‘default’ regulation conditions, to exclude the threshold regulation conditions (‘noThresholds’), etc. ii) whether you want to use synchronous or asynchronous (‘async’) updates. iii) The maximum path length to be considered. Comments can be written following ‘//’, but cannot include punctuation. See [here](#runningsolver) for setting these in the solver directly. Next, define the components and regulation conditions they can take. Component names are separated by semi-colons, while regulation conditions are positioned within round brackets and individually separated by a comma. A sequence of regulation conditions can be indicated by ‘..‘ between the start and end of the sequence. Next define the interactions between the components. Interactions appear on separate lines and are ordered according to ‘source’, ‘target’, either ‘positive’ or ‘negative’, and finally an ‘optional’ argument, which if not used, defines the interaction to be definite. These arguments can be tab-separated, or simply space-separated. Each interaction must be completed with a semi-colon. | // Settingsdirective regulation noThresholds;directive updates sync;directive length 20;// ComponentsS1(0); S2(0); // SignalsA(0..8); B(0..8); C(1,3,5); // TFs// Definite interactionsS1 S1 positive;S2 S2 positive;S1 A positive;S2 B positive;// Possible interactionsA C positive optional;A B positive optional;B A positive optional;B C positive optional; |

Table : How to encode a set of components and interactions, illustrated using our toy example.

### Defining an external signal

You may wish to define an external signal that can interact with the other components in your network, but is not itself regulated by one of these components. To do so, define this signal as a component as normal, and also define any outgoing interactions. To prevent a signal from switching on or off arbitrarily, in addition, you should also add a self-activation interaction. E.g. For the signal S, you would add the following definite interaction:

S S positive;

Whether a signal is on or off for a given experiment must be defined at the first timestep (see [Section](#observations) below).

### Components that need activators

If you wish to restrict the topology of a network such that a given component must have an activator, then include ‘[!]’ immediately following the component name. E.g. A[!](0..8); B[!](0..8); C[!](1,3,5); requires that these three components have activating interactions.

### Visualising the set of interactions

The ABN that has been encoded is visualised in the ‘Network’ tab on the left hand side. The set of components and interactions will appear, with positive interactions drawn as arrows, and negative interactions drawn as T-shaped lines. Depending on the browser you are using, optional interactions appear either as solid or dashed, but always grey lines, while definite interactions are solid black lines. If you edit the set of components and interactions, the graphical representation is automatically refreshed.

For those genes that can be knocked down (see [Section](#GeneticPerturbtions) below), the component will appear as coloured red. For those genes that can be overexpressed, the component will appear as coloured green. For those genes that can be either knocked down or overexpressed, the component will appear as coloured yellow.

# Encoding a Set of Experimental Observations

The user next defines the set of experimental observations, which are used to constrain the set of possible models that are defined by the ABN.

### Manually defining experimental observations

The set of experimental observations are written in the ‘Observations’ tab. As with components and interactions, they can be entered manually or loaded from the prewritten ‘.spec’ file.

Each network is evolved as a transition system, with either asynchronous or synchronous updates. Typical experimental observations place restrictions on the states that should be attained at different timesteps, e.g. initial and final gene expression patterns.

For example, a typical experiment could be written as shown in Table 2. The colour of the text shows how it will appear in RE:IN. This allows you to easily navigate files as they increase in length (for example, if you double click on ‘$Conditions1’, coloured boxes will appear on the scroll bar to indicate where this string appears throughout the file, allowing you to quickly jump to its definition.)

Individual experiments are marked with ‘#’, while state values collected under a predicate are marked with ‘$’. Additional experiments can be added, for example using the tags, ‘#Experiment3’, ‘#Experiment4’, and predicates, such as ‘$Conditions1’, can be re-used in different experiments.

While the example in Table 2 gives both initial and final states in the two experiments, this is not a requirement of the solver. If an initial condition is not prescribed for a given experiment, the solver will search for all models that can reach the specified final state, and will choose an initial condition that allows this observation to be satisfied. Similarly, the state of each component *does not* need to be specified at each timestep, if it is not known. In these cases, the solver will satisfy what is specified, and identify states for the remaining gene appropriately.

Note that in the example, we specify that the trajectory must reach a given state at time step 18, and that this be a fixed point. This requirement defines that the trajectory stabilises, given that they are deterministic and have synchronous updates.

|  |
| --- |
| // Observation predicates allow you to collect and tag gene states$Conditions1 := { S1 = 0 and S2 = 1};$Conditions2 := { S1 = 1 and S2 = 1};$Expression1 := {A = 1 and B = 1 and C = 1};$Expression2 := {A = 0 and B = 1 and C = 1};// Observations define the state you expect the system to be in at different time steps#Experiment1[0] |= $Conditions1 and#Experiment1[0] |= $Expression1 and#Experiment1[18] |= $Expression2 andfixpoint(#Experiment1[18]); // We expect to stabilise in this state#Experiment2[0] |= $Conditions2 and#Experiment2[0] |= $Expression2 and#Experiment2[18] |= $Expression1 andfixpoint(#Experiment2[18]); |

Table : Defining the set of experimental constraints.

### Saving experimental observations

The set of experimental observations can easily be saved using the ‘Save’ button. Observation files should be appended with ‘.spec’ but the solver will accept ‘.txt’ files. (As with the component / interaction files, experimental observations can also be written in any text editor and saved accordingly.)

### Loading experimental observations

Load a pre-prepared set of experimental observations using the ‘Load’ button.

### Model Diameter / Setting Timesteps

RE:IN requires that the time steps be specified in individual experiments. First, it is important to note that these values do not need to match precisely with physical time. For example, when specifying that a steady state should be reached ‘eventually’, selecting a large final time step also allows such experiments to be reproduced with shorter trajectories, where the final steady state is repeated a number of times. Similarly, when encoding transient states or cycles, the precise step of each event can be omitted by instead specifying a range (e.g. the event is satisfied at step 0 or at step 1, etc.).

In principle, the recurrence diameter of a system provides an over-approximation of the longest loop-free trajectories that need to be considered to reproduce any experiment. We have implemented a naïve approach in RE:IN to allow users to both check and find the recurrence diameter for a given ABN. Even so, the computation of the exact diameter is not always necessary and any value that is provably larger than the diameter can be used instead.

With a set of models encoded, navigate to the Tools tab. The ‘Check Diameter’ button can be used to test whether a value provided in the box on the right hand side is sufficient. A pop up dialog box will report the results of the test. The ‘Find Diameter’ button can be used to ask RE:IN to identify the necessary trajectory length to use in experiments. However, this functionality should be used with caution, as it cannot be used in certain scenarios (e.g. asynchronous update schemes) and may take a significant time to run.

If you check the box labelled, ‘Consider observations as part of diameter computation’, then the diameter computation will find the longest loop-free path while taking account of any experimental observations you have encoded already.

# Finding Models that Satisfy Experimental Observations

### Setting Analysis Options

The ‘Options’ tab allows you to enforce certain settings on the analysis.

**Solutions limit**: The maximum number of models you wish to enumerate.

**Interactions limit**: Use to restrict the number of optional interactions that comprise any given model solution (this excludes the definite interactions). The default is set to 0, which means that there is no limit imposed. However, to look for models with only 4 interactions, you would replace this with a ‘4’.

**Regulation Conditions:** Use to restrict the set of regulation conditions to consider. ‘Default’ corresponds to the entire set shown in Fig. 1, together with the two threshold rules. ‘No threshold rules’ removes conditions 18 and 19. ‘Legacy’ implements the set of 18 regulation conditions defined in the first version of RE:IN (*3*).

**Updates:** Use to set either asynchronous or synchronous updates.

**Unique solutions:** You can enumerate concrete models that are unique in topology (‘Interactions only’), unique in either topology or regulation conditions assigned to each component (‘Interactions and regulation conditions’), or unique in the trajectories required to satisfy the different experiments (‘Interactions, regulation conditions and experiments’).

**Experiment length:** This option sets the maximum trajectory length to be allowed for each experiment.

### Running the solver

### Once the set of interactions and observations have been encoded, and options set, to identify and enumerate only those models capable of satisfying the observations, click the ‘Run Analysis’ button at the top of the left-hand pane.

If model solutions exist, then the model summary will appear over time in the ‘Results’ tab in the right panel. Once the solver has found all of the models that can satisfy the experimental observations, a dialog box will appear, saying ‘No additional solutions found’ if you have yet to reach the maximum solution limit, or ‘Solution limit reached’ if you have attained the maximum solution limit you set. In the latter case, more solutions may exist.

### Visualising model solutions

In the model summary, each valid model that satisfies all of the experimental observations appears as a numbered column of red and green squares. Each row of the column corresponds to an interaction (described on the far left), with a green square indicating that the interaction is positive, and a red square indicating that the interaction is negative.

The first column is a summary – the brightness of the red and green squares correlates with the incidence of this interaction across all solutions found thus far.

The definite interactions that were defined do not appear in this list, as it summarises only the optional interactions that are instantiated in each solution.

### Visualising solution topologies

To visualise the topology of a specific model solution, navigate to the ‘Solution Browser’ tab. Select the solution you wish to visualise from the drop-down menu.

With the ‘Interaction Graph’ tab selected, you can see a visualisation of the chosen solution. The red arrows and T-shaped lines correspond to the optional interactions that have been instantiated in this solution, while the definite interactions remain coloured black.

The numbers in round brackets next to the component names correspond to the specific regulation condition that has been assigned to that component, for that model, which correspond to the numbers defined in Fig. 1.

### Visualising model trajectories

With a specific model chosen, you can visualise specific experiments in the ‘Experiment Visualization’ tab. This plots the trajectories of the individual components for the experiment selected in the drop down menu. The x-axis corresponds to time steps, and the y-axis to the expression level of each gene (which is either 0 or 1).

Trajectories can also be viewed as a table of Boolean values in the ‘Experiment Data’ tab. Here, the rows correspond to timesteps, and columns to individual components.

# When the Solver Cannot Find Any Models

It is possible that there are no concrete models that can satisfy the set of experimental constraints. In this event, a dialog box will appear saying, ‘No solutions found.’ This may be because you are missing key interactions between certain components, or because the experimental observations are incorrect, or incompatible.

Under these circumstances, it is worth checking both the set of interactions and experimental observations for simple errors. If you are certain of the observations, it may be that you are missing vital interactions. This is where the usefulness of this approach is most evident: by adding in optional interactions between components that were not previously considered and re-running the solver, you can identify new interactions that might be needed to satisfy your observations. Such predicted interactions could then be investigated experimentally. Examples of this are given in our methods paper (*1*).

# Required and Disallowed Interactions

If you have identified a number of model solutions that satisfy the set of experimental observations, it is interesting to uncover interactions that are common to all models (required), or will never appear in any model solution (disallowed). While the green/red results summary panel can be used to identify such interactions in cases where there is a small solution set, it is more challenging if you have a large set of components and interactions, and potentially billions (or billions of billions…) of solutions.

To identify the set of required and disallowed interactions, load up the components, interactions and constraints as before, but now click the ‘Find Required Interactions’ button at the top.

Once the solver has verified whether each of the optional interactions is sometimes required, definitely required, or never required, a dialog box will appear to confirm that the analysis is complete, and this set will be available to examine in the ‘Notes’ tab. The set of interactions in the network visualisation tab will have updated accordingly: optional interactions that are required have changed from grey to black, and any optional interactions that are never required will have disappeared. In the toy example we consider, the positive interaction from gene B to gene C is required.

# Minimal Models

RE:IN also enables the user to examine minimal networks, which are those with the fewest optional interactions instantiated that satisfy experimental observations.

To identify the set of minimal models, load up the components, interactions and constraints as before, but now click the ‘Find Minimal Models’ button. The solver will populate the Results tab with these models, and the pop up dialog box will indicate the number of interactions in each.

# Experiments with Genetic Perturbations

It is possible to define experiments in which one or more genes are knocked out (KO), or overexpressed (FE). If you require this, when you define a component, check the ‘KO’ or ‘FE’ box appropriately in the interactions editor. If you wish to define this within a .net file, immediately append a ‘-‘ for KO, and / or a ‘+’ for FE within square brackets, just before the round brackets that define the allowed regulation conditions for that component.

For example,

|  |
| --- |
| A[+-](0..8); B[-](0..15); C(1,2,3); D[+](0..5); |

Above, A can be both knocked out and overexpressed, B can be knocked out and D can be overexpressed. C cannot be perturbed at all. When you have defined both knock outs and overexpressions, the network layout will update accordingly. Genes that can be knocked out are coloured red, genes that can be overexpressed are green, and genes that can be either are yellow.

It is vital that the experimental constraints account for this additional flexibility. Any genes that are defined as having the capacity to be perturbed, must be defined as knocked out / not knocked out, or overexpressed / not overexpressed in *each* observation. This is because RE:IN now has the flexibility to perturb those genes, unless told otherwise.

An example of how to encode observations appropriately in this instance is as follows. In this example, in experiment 3, we want to knockout gene B, and not overexpress any genes. Therefore, we need to state explicitly that there are no knockouts or overexpressions in the other experiments. These definitions will hold true from the time point at which they are defined.

The solver recognises whether a gene is knocked out with ‘KO()’, and whether a gene is overexpressed by ‘FE()’. Therefore, ‘KO(A)=1” corresponds to A being knocked out, while ‘KO(B)=0’ means that B is not. Likewise, ‘FE(D)=1’ means that D is overexpressed.

|  |
| --- |
| // Observation predicates allow you to collect and tag gene states$Conditions1 := {S1 = 0 and S2 = 1};$Conditions2 := {S1 = 1 and S2 = 1};$Expression1 := {A = 1 and B = 1 and C = 1};$Expression2 := {A = 0 and B = 1 and C = 1};$Expression3 := {A = 1 and B = 0 and C = 1};$Expression4 := {A = 0 and B = 0 and C = 0};// Define possible knock outs$NoKnockouts := {KO(A) = 0 and KO(B) = 0};$BKnockedOut := {KO(A) = 0 and KO(B) = 1};// Define possible overexpressions$NoOverexpressions := {FE(A) = 0 and FE(B) = 0};// Observations define the state you expect the system to be in at different time steps#Experiment1[0] |= $Conditions1 and#Experiment1[0] |= $Expression1 and#Experiment1[0] |= $NoKnockouts and#Experiment1[0] |= $NoOverexpressions and#Experiment1[18] |= $Expression2 andfixpoint(#Experiment1[18]); // We expect to stabilise in this state#Experiment2[0] |= $Conditions2 and#Experiment2[0] |= $Expression2 and#Experiment2[0] |= $NoKnockouts and#Experiment2[0] |= $NoOverexpressions and#Experiment2[18] |= $Expression1 andfixpoint(#Experiment2[18]);// Experiment where B is knocked out#Experiment3[0] |= $Conditions1 and#Experiment3[0] |= $Expression3 and#Experiment3[0] |= $BKnockedOut and#Experiment3[0] |= $NoOverexpressions and#Experiment3[18] |= $Expression4 andfixpoint(#Experiment3[18]); |

# Making New Predictions

Once you have identified that there exists a set of models that satisfy the set of experimental constraints you have defined, you can explore the new behaviour this set of models predicts. For example, you might want to investigate the effect of perturbing certain genes on the remaining gene expression pattern. If you have more than one model solution, then predictions must be those that are made by *all* of the solutions, and not just one that you have randomly chosen – in this way, you predict behaviour that is supported by all of your experimental observations in an unbiased manner.

Consider the following example. We want to investigate whether the simple set of models that have been derived thus far predicts that gene A will be expressed if gene B is knocked out. To examine this, we construct candidate experimental constraints, and test the set of models against these new constraints separately:

1. A is expressed when gene B is knocked out.
2. A is not expressed when gene B is knocked out.

First, we test (1). If we find solutions under this constraint, this does not reveal whether all of the models satisfy this constraint. Therefore we next test (2), independently. If we find solutions to this constraint, then our results are inconclusive and we cannot form a prediction of the response of the network, because a subset of the solutions lead to expression of A, while a different subset lead to loss of expression of A. If we do not find solutions to (2), then we predict that expression of A is sustained. Likewise, if there are no solutions that satisfy (1), but solutions are found for (2), then we predict that A is never expressed when B is knocked out.

If you are concerned with the expression of more than one gene under genetic perturbation, then you can define this predicate using a unique tag (e.g. $ExpectedFinalExpressionPattern), and encode the following as candidate experimental constraints to be tested separately as above:

1. #PredictionTesting[10] |= $ExpectedFinalExpressionPattern
2. not(#PredictionTesting [10] |= $ExpectedFinalExpressionPattern)

‘not’ is used here to refer to any state that is not that defined by $ExpectedGeneExpressionPattern.

# Example Files

By clicking on the links provided on the RE:IN webpage, you can quickly access pre-loaded examples that correspond to the case studies we present in our publications (*1*, *3*).

# References

1. B. Yordanov *et al.*, A Method to Identify and Analyze Biological Programs through Automated Reasoning. *Under Rev.* (2016).

2. F. Li, T. Long, Y. Lu, Q. Ouyang, C. Tang, The yeast cell-cycle network is robustly designed. *PNAS*. **101**, 4781–6 (2004).

3. S.-J. Dunn, G. Martello, B. Yordanov, S. Emmott, A. G. Smith, Defining an essential transcription factor program for naive pluripotency. *Science (80-. ).* **344**, 1156–1160 (2014).