# Visual DSD User Manual

Carlo Spaccasassi[1], Neil Dalchau[1], Matthew Lakin[1],
Rasmus Petersen[1], and Andrew Phillips[1*]

[1]Microsoft Research, Cambridge CB1 2FB, UK
[*]andrew.phillips@microsoft.com

March 3, 2019

## Contents

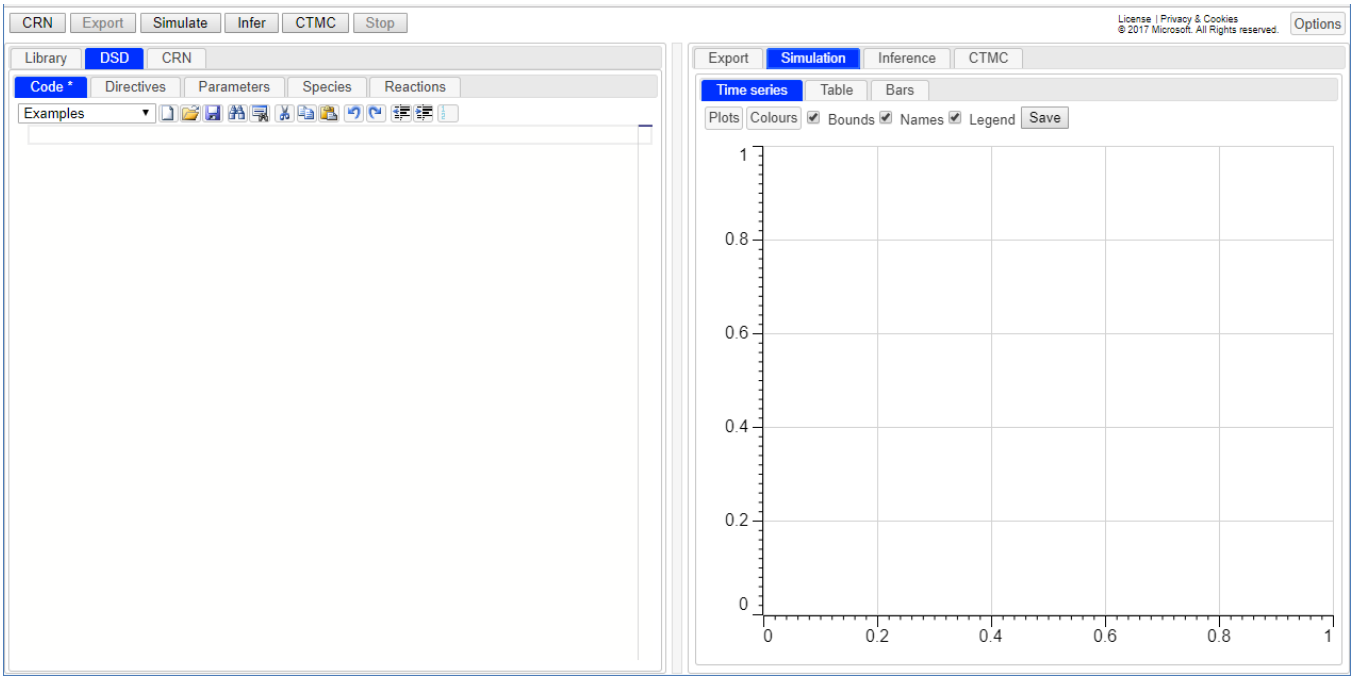



**Figure 1: The Visual DSD user interface.**

# 1 Introduction

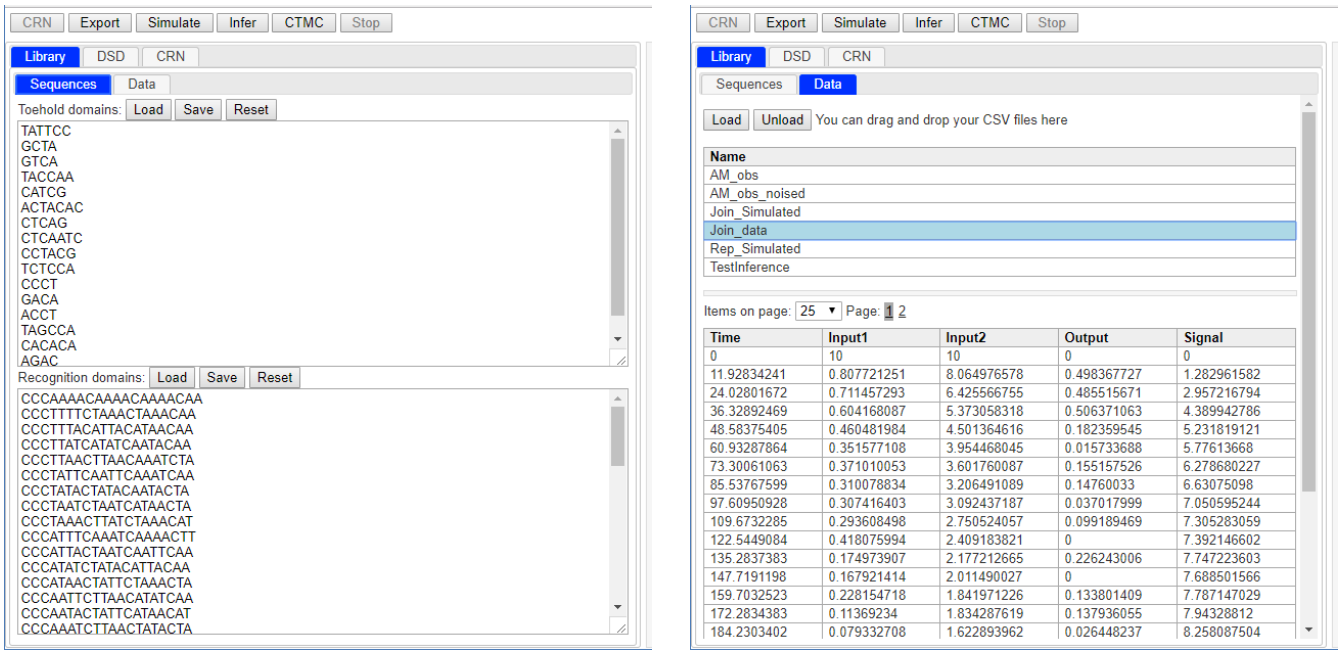
Visual DSD is a programming language and software tool for designing computational devices made of DNA. It was originally proposed in [1] and has since been extended with a broad range of modelling and analysis capabilities [2–14]. The main computational mechanism that underpins Visual DSD is DNA strand displacement [15], whose systematic use was pioneered in [16]. DNA strand displacement can implement a broad range of computation, including any computation that can be expressed as a chemical reaction network [17]. It involves an invading single strand of DNA displacing an incumbent strand hybridized to a template strand. The process is mediated by a short, single-stranded region of DNA referred to as a *toehold*. We assume that the reader is familiar with the basics of DNA strand displacement. Additional technical details on the Visual DSD programming language can be found in [6].

Visual DSD compiles a collection of DNA molecules into a chemical reaction network (CRN). It also includes a stochastic simulator, which computes a possible trajectory of the system and plots the populations of species over time, together with a deterministic simulator, which forms and solves an Ordinary Differential Equation (ODE) representation of the dynamics of the system. The reachable state space of the system can also be constructed as a continuous time Markov chain (CTMC).

Visual DSD is available as a web-based graphical application, which runs entirely in a web browser on the user's device. Information entered into the tool by the user remains on the user's device and is not transmitted to the server. Visual DSD is compatible with most modern web browsers, and has been tested on recent versions of Edge, Firefox, Safari and Chrome. It is also compatible with smart phones and tablets running the Chrome browser, though it has not been customised for these devices. To access the user interface, browse to https://classicdsd.azurewebsites.net. The tool should load within a few seconds. Note that when a new version of the tool is released online, the web page may need to be refreshed in order to access the latest version. A downloadable version is also available from https://dsd.azurewebsites.net/server.

# 2 User Interface

A screen shot of the Visual DSD user interface is shown in Figure 1. The top right contains a link to the License agreement, together with an Options menu for customising the user interface. The top left displays a row of buttons that perform actions on user inputs, referred to as *Action buttons*. The tabs on the left display inputs that are edited by the user, referred to as *Input tabs*. They follow a left-to-right progression, such that information is propagated between tabs from left to right. The tabs on the right display outputs that are generated when an Action button is pressed, referred to as *Output tabs*. We refer to specific tabs by their name and to nested tabs by their name preceded by the name of their parent, for example the DSD/Code tab. In this section we illustrate the main functionality of the user interface tabs, using a running example.




(a) Library/Sequences tab

(b) Library/Data tab

**Figure 2: The Library tab.**

## 2.1 Library

The *Library* tab (Figure 2) contains DNA sequences and experimental data that are used as inputs to a Visual DSD program. These stored in two nested tabs, described below.

### 2.1.1 Library/Sequences

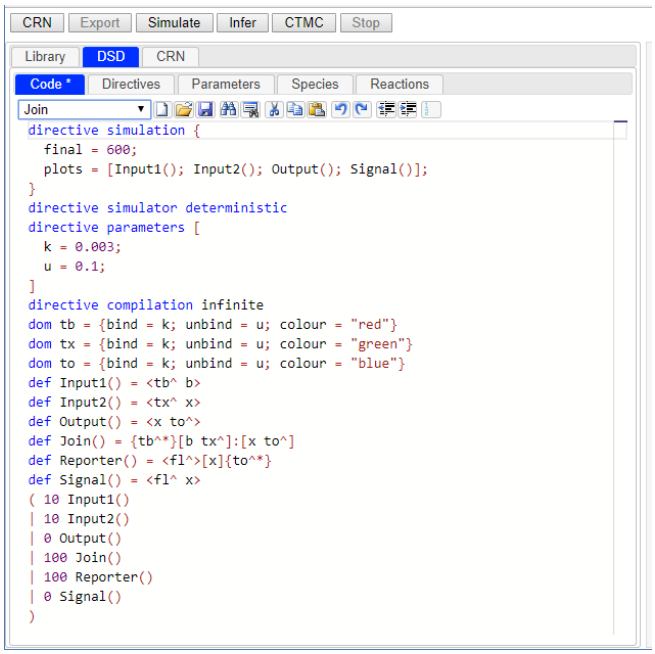
The *Sequences* tab (Figure 2a) contains two lists of DNA sequences, which are used to assign a default DNA sequence to each domain in a given Visual DSD program. The first list specifies default sequences for toehold domains, which are assumed to be short enough such that they bind reversibly to their complementary sequence. The second list specifies default sequences for *recognition* domains, which are assumed to be long enough such that they bind irreversibly to their complementary sequence. The list of recognition domains was kindly provided by Erik Winfree and Lulu Qian, based on sequences published in [18]. Both lists can be edited by the user.

### 2.1.2 Library/Data

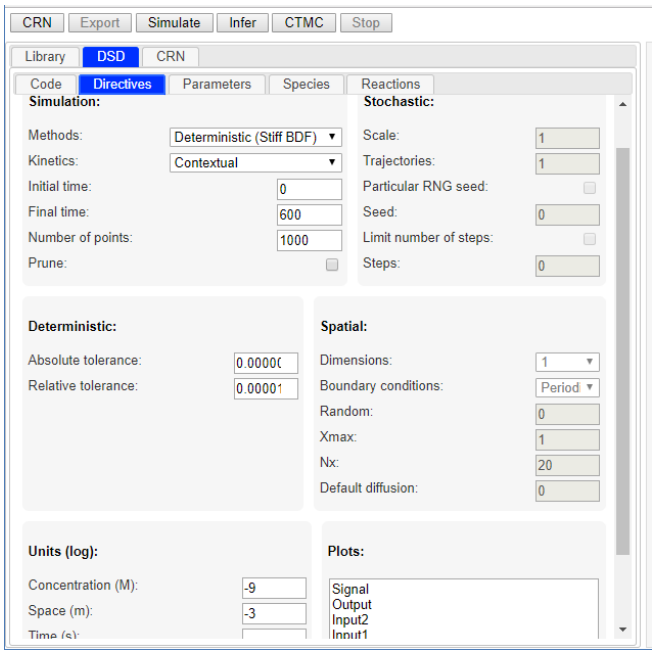
The *Data* tab (Figure 2b) contains tables of experimental data, which can be compared against model simulations in order to infer model parameters. A number of tables are provided by default, and additional tables can be added by the user, in either comma separated values (csv) format or tab separated values (tsv) format. To load a table, either click the Load button to open a file navigation window and browse to the file, or drag the file onto the list of existing tables. The table is added to the list using the name of the file with the extension removed, and can then be referenced by this name within a Visual DSD program. Each file loaded by the user must conform to a standard format. It should contain a single row of headers, which are used as labels for the various columns but do not affect the program behaviour. Subsequent rows contain data points. The first column is reserved for specifying the time at which each row of data was collected. Each file is assumed to contain a single time column.

## 2.2 DSD

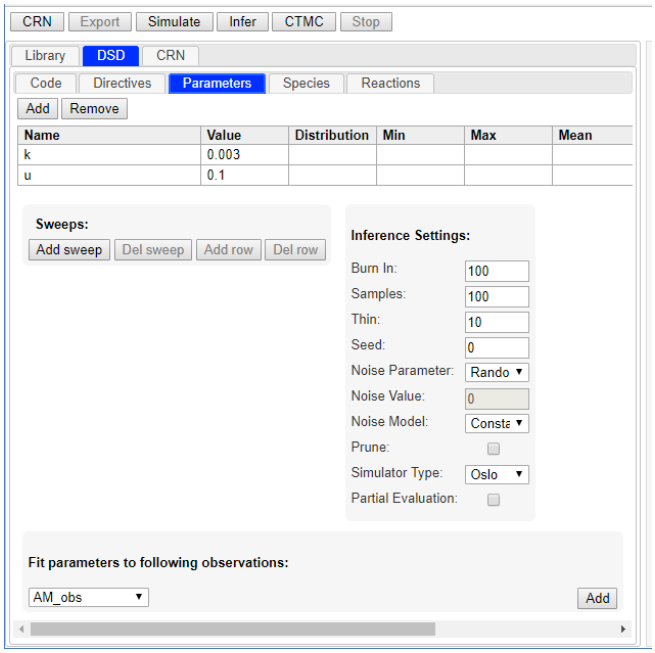
The *DSD* tab (Figure 3) contains five nested tabs that follow a left-to-right progression, such that the contents of the Directives, Parameters, Species and Reaction tabs are obtained from the contents of the Code tab. When one of the nested tabs is modified by the user, an asterisk is displayed next to the tab name. Below we briefly describe the contents of the nested tabs for our running example, where the first four nested tabs are displayed in Figure 3. The Reactions tab is omitted, since it is empty in this example.



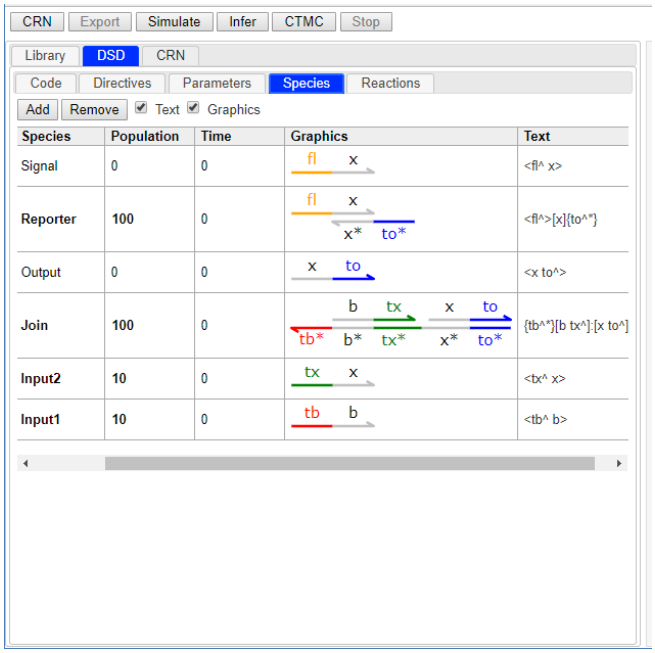

(a) DSD/Code tab




(b) DSD/Directives tab




(c) DSD/Parameters tab




(d) DSD/Species tab

**Figure 3: The DSD tab.** (a) The code for this example is provided in the Examples menu of the DSD/Code tab, under the heading Manual/Join. This is used as a running example for the remainder of this document.

### 2.2.1 DSD/Code

The *Code* tab (Figure 3a) contains a textual editor for Visual DSD programs. Standard text editing functionality is provided, including syntax highlighting, loading and saving of files, find and replace, copy and paste, undo and redo, and display of line numbers. The font size of the editor is adjusted by holding the control (Ctrl) key while scrolling with a mouse or other input device. Several example programs are also provided in the *Examples* drop-down menu. The full syntax of Visual DSD programs is summarised in Section 4. For illustration, we briefly describe the *Join* program from the *Manual* section of the *Examples* menu (Figure 3a). The code for this program can be divided into four sections: *directives*, *domains*, *modules* and *initial conditions*:

**Directives** are defined using the `directive` keyword and determine how the DSD program should be executed. The `simulation` directive determines how the program should be simulated, where `final = 600` sets the final simulation time to 600, and `plots = [Input1(); Input2(); Output(); Signal()]` defines the DNA

species to be plotted during simulation. Since no time units are specified explicitly, seconds (s) are used by default. The `simulator` directive defines the type of simulator to be used. In this example the simulator is set to `deterministic`, which means that the program will be compiled to a set of ordinary differential equations and simulated deterministically by numerical integration. The `parameters` directive defines the global parameters of the program, which are assumed to be floating point numbers. In this example there are two parameters, `k = 0.003` and `u = 0.1`. The `compilation` directive defines the method to be used when compiling a DSD program to a chemical reaction network. In this example, the `infinite` compilation method is used, which assumes that the concentration of DNA species is low enough such that toehold unbinding and branch migration happen significantly faster than binding and are assumed to be effectively infinite. Additional details on compilation methods are provided in Section 4.3.1.

**Domains** are defined using the `dom` keyword, where each domain is assumed to represent a unique DNA sequence. We assume that a domain can bind to its complement but cannot interact with any other domains in the system. As a result, care must be taken to ensure that non-interfering DNA sequences are used when performing the corresponding biological experiments, for instance by relying on appropriate DNA coding strategies [18]. The code `dom tb = {bind = k; unbind = u; colour = "red"}` defines a domain `tb` with binding rate `k`, unbinding rate `u`, and colour `red`. The domains `tx` and `to` are defined in a similar fashion. Binding rates are assumed to have units of concentration$^{-1}$time$^{-1}$, while unbinding rates are assumed to have units of time$^{-1}$. The default concentration units are nanomolar (nM), while the default time units are seconds (s). If no rates are specified then default rates are used, and if no colour is specified then each toehold domain is assigned a distinct colour. If a domain is not declared using the `dom` keyword then default domain properties are used.

**Modules** are defined using the `def` keyword, where a given module can have zero or more arguments enclosed in parentheses. The code `def Input1() = <tb^ b>` defines a module `Input1` with no arguments, equal to the DNA strand `<tb^ b>`. The modules `Input2`, `Output` and `Signal` are defined in a similar fashion. The code `def Join() = {tb^*}[b tx^]:[x to^]` defines a module `Join` with no arguments, equal to the DNA complex `{tb^*}[b tx^]:[x to^]`. The `Reporter` module is defined in a similar fashion. The syntax of DNA strands and complexes for this example is briefly described in Section 2.2.4.

**Initial conditions** are defined by initialising one or more DNA species, separated by the parallel composition operator (`|`). The code `100 Input1()` initialises the species `Input1` to `100`. Since the simulator is set to deterministic, the number `100` represents a concentration, with default units of nM. If the simulator is set to stochastic then this number represents a population of individuals.

### 2.2.2 DSD/Directives

The *Directives* tab (Figure 3b) displays a table representation of the program directives from the Code tab, with the exception of directives related to parameters. In this example, when the user selects the Directives tab, the `simulation` and `simulator` directives are parsed from the Code tab (Figure 3a) and displayed in table form. Default values are also displayed for those directives that are not explicitly declared in the program. For example, the default initial time of the simulation is displayed as 0 and the default number of points for plotting the simulation results is displayed as 1000. Note that the table display of the `compilation` directive is not yet supported.

### 2.2.3 DSD/Parameters

The *Parameters* tab (Figure 3c) displays a table representation of the program directives that relate to parameters. In this example, when the user selects the Parameters tab, the `parameters` directive is parsed from the program and displayed in table form. Default values are also displayed for those parameter directives that are not explicitly declared in the program.

### 2.2.4 DSD/Species

The *Species* tab (Figure 3d) displays a table representation of the DNA species from the program. Definitions for domains and modules are inlined, resulting in a list of DNA species with their associated attributes, including the species name, population, initialisation time, and corresponding graphical representation. In this example, the species name is obtained from the name of the module in which it is defined. There are two types of species in this example, *strands* and *complexes*:

**DNA strands** are represented as a sequence of domains enclosed in angle brackets, where the 3′ end of the strand is assumed to be on the right. Toehold domains are represented by appending the (`^`) character to the domain name. For example, `<tb^ b>` represents a strand consisting of the toehold domain `tb^` followed by the domain `b`, defined in Figure 3d as the `Input1` strand. Note that DNA strands can also be represented as a sequence of domains enclosed in curly brackets, where the 3′ end of the strand is instead assumed to be on the left.

For example, the strand `<tb^ b>` can also be written as `{b tb^}`. This is because strands are identical up to *rotation symmetry*, such that we can write the same strand either from left to right or from right to left.

**DNA complexes** are represented as a sequence of *segments*, where each segment is either a DNA strand as defined above, or a double stranded duplex with overhanging or underhanging strands to the left or right. Complementary domains are represented by appending the (`*`) character to the domain name. For example, the code `{tb^*}[b tx^]:[x to^]` represents a complex consisting of two segments, defined in Figure 3d as the `Join` complex. The first segment `{tb^*}[b tx^]` represents a double stranded duplex consisting of the strand `<b tx^>` bound to its complementary strand `{b* tx^*}`, with an additional single stranded underhang `{tb^*}` to the left of the duplex. The second segment `[x to^]` represents a double stranded duplex consisting of the strand `<x to^>` bound to its complementary strand `{x* to^*}`. These two segments are joined together along the bottom strand by the segment connection operator (`:`). Note that although the textual representation of complexes is defined as a connection of segments, in reality the connection results in a single continuous bottom strand, as can be seen by the graphical representation of the `Join` complex (Figure 3d). The `Reporter` complex is represented in a similar fashion.

#### 2.2.5 DSD/Reactions

The *Reactions* tab displays a table representation of the DNA reactions defined in the program. In this example the Reactions tab is empty, since there are no reactions defined explicitly in the program, so the tab is not displayed in Figure 3.

### 2.3 CRN

The *CRN* tab (Figure 4) is populated by pressing the *CRN* button when the DSD tab is selected. This parses the DSD program to produce a set of initial DNA species, which is then processed using a built-in *reaction enumerator* to generate the corresponding chemical reaction network. Additional details about the reaction enumerator are described in [6]. The generated chemical reaction network is then used to populate all of the nested CRN tabs, which follow a left-to-right progression similar to the nested DSD tabs. Below we briefly describe the contents of the nested CRN tabs for our running example. The Code, Species and Reactions tabs are displayed in Figure 4. We omit the Directives and Parameters tabs, since in this example they are similar to the corresponding nested DSD tabs of the same name.

#### 2.3.1 CRN/Code

The *Code* tab (Figure 4a) is a textual editor for Visual CRN programs, and has the same functionality as the DSD/-Code tab. The full syntax of Visual CRN programs is summarised in Section 3. For illustration, we briefly describe the CRN program generated from our running example (Figure 4a), which consists of three sections: *directives*, *initial conditions* and *reactions*:

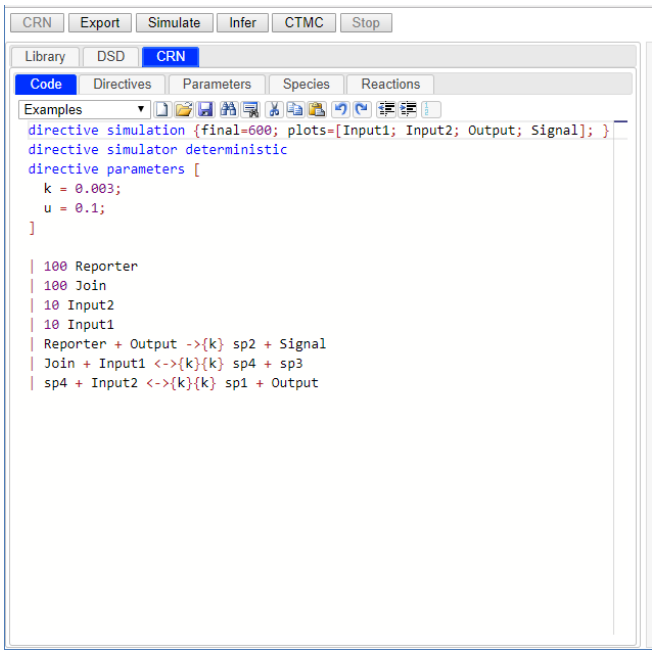
**Directives** are defined using the `directive` keyword and determine how the CRN program should be executed. Directives that are specific to the DSD program are not included in the generated CRN program, since they are not applicable to a CRN. For the running example, the `compilation` directive from the DSD program is not included.

**Initial conditions** are defined by initialising one or more CRN species, separated by the parallel composition operator (`|`). For example, the code `100 Input1` initialises the species `Input1` to 100.

**Reactions** are defined by declaring one or more reactant species, one or more product species and a reaction rate. Reactions are separated by the parallel composition operator (`|`). The reaction `Reporter + Output ->{k} sp2 + Signal` defines a reaction with two reactants, `Reporter` and `Output`, two products, `sp2` and `Signal`, and rate `k`. Note that the `Reporter`, `Output` and `Signal` species are present in the original DSD program, however the species `sp2` was generated during reaction enumeration, along with species `sp1`, `sp3` and `sp4`. The reaction `Join + Input1 <->{k}{k} sp4 + sp3` defines a reversible reaction, where the forward and reverse reaction both take place at rate `k`. Note that species and reaction definitions can be interleaved in the CRN program using the parallel composition operator (`|`), but are here written in two separate groups for clarity.

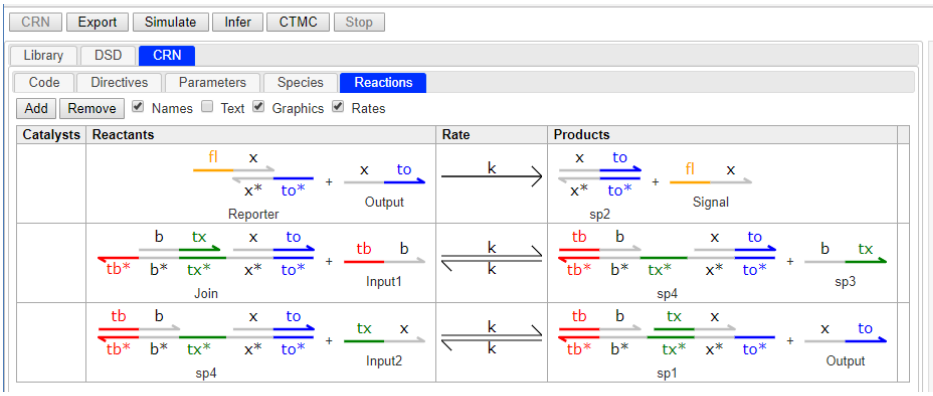
#### 2.3.2 CRN/Species

The *Species* tab (Figure 4b) displays a table representation of the CRN species present in the CRN program. This includes all of the species present in the original DSD program, together with the new species `sp1`,...,`sp4` generated during reaction enumeration.




(a) CRN/Code tab


(b) CRN/Species tab

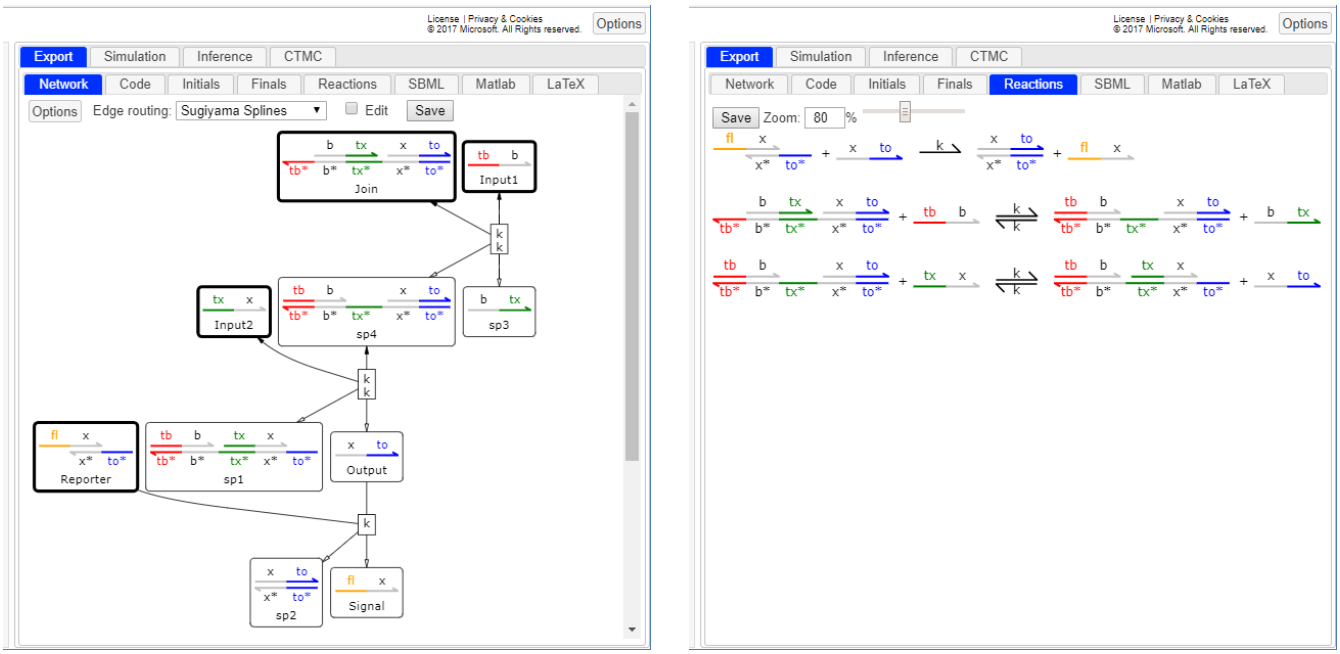



(c) CRN/Reactions tab

**Figure 4: The CRN tab.**

### 2.3.3 CRN/Reactions

The *Reactions* tab (Figure 4c) displays a table representation of the chemical reactions present in the CRN program. Species are represented graphically, consistent with their representation in the Species tab. We briefly summarise how the reactions in this tab were enumerated for the running example, and refer the reader to [6] for a more detailed description of the reaction enumeration algorithm. The first reaction in the Reactions table corresponds to `Reporter + Output ->{k} sp2 + Signal`. In this reaction, the `to^` domain of the `Output` strand binds to the `to^*` domain of the `Reporter` complex and displaces the `Signal` strand. This also results in a new species `sp2`, which corresponds to the duplex `[x to^]`. The reaction occurs at rate `k`, which is the rate associated with the domain `to`, as defined in the DSD program. The second reaction in the table corresponds to `Join + Input1 <->{k}{k} sp4 + sp3`. In this reaction, the `tb^` domain of the `Input1` strand binds to the `tb^*` domain of the `Join` complex and displaces the `sp3` strand to produce the `sp4` complex. The displacement reaction takes place at rate `k`, which is the rate associated with the domain `tb`, as defined in the DSD program. Note that the reaction is reversible, since the `tx^` domain of the `sp3` strand can in turn bind to the `tx^*` domain of the `sp4` complex, to displace the `Input1` strand and restore the `Join` complex. This reverse reaction takes place at rate `k`, which is the rate associated with the domain `tx`. Similarly, the `Input2` strand can bind the `sp4` complex to displace the `Output` strand. Together, these two reactions define a Join operation, where the `Output` strand is produced only if both `Input1` and `Input2` are present. The `Output` strand then displaces a `Signal` strand, which is assumed to be measurable experimentally. This is represented by and additional `fl` domain, which represents a fluorescent molecule attached to the end of the `Signal` strand and is used quantify the amount of `Signal` being produced via fluorescence.




(a) Export/Network tab

(b) Export/Reactions tab

**Figure 5: The Export tab.**

## 2.4 Export

The *Export* tab (Figure 5) is populated by pressing the *Export* button when either the DSD or CRN tab is selected. This parses the DSD or CRN program, and produces a range of graphical and textual formats that can be saved as files. The Export tab is also automatically populated by pressing the CRN button when the DSD tab is selected. We briefly summarise the contents of the Export tab for the running example when the DSD/Code tab is selected and the CRN button is pressed:

**Export/Network** displays a Network representation of the CRN/Reactions tab (Figure 5a). The network consists of two types of nodes, representing species and reactions. Each species node contains a DNA species from the CRN/Species tab, together with the name of the species. For species with a non-zero initial population, the node is highlighted with a bold outline. Each reaction node contains either a single rate, denoting an irreversible reaction, or two rates, denoting a reversible reaction, with the forward rate on top and the reverse rate on the bottom. Edges with an empty arrowhead from a reaction node to a species node denote the products of a reaction, while edges with no arrowhead denote the reactants. If the reaction is reversible, a solid arrowhead is used to denote the reactants. For example, the reaction represented in the top right of the network in Figure 5a corresponds to the reversible reaction `Join + Input1 <->{k}{k} sp4 + sp3`.

**Export/Code** displays an automatically formatted version of the CRN/Code tab.

**Export/Initials** displays the contents of the CRN/Species tab in a format that can be saved as a scalable vector graphics (SVG) file.

**Export/Finals** displays the DNA species present at the end of a simulation, though this feature is still under development and currently only supports stochastic simulation.

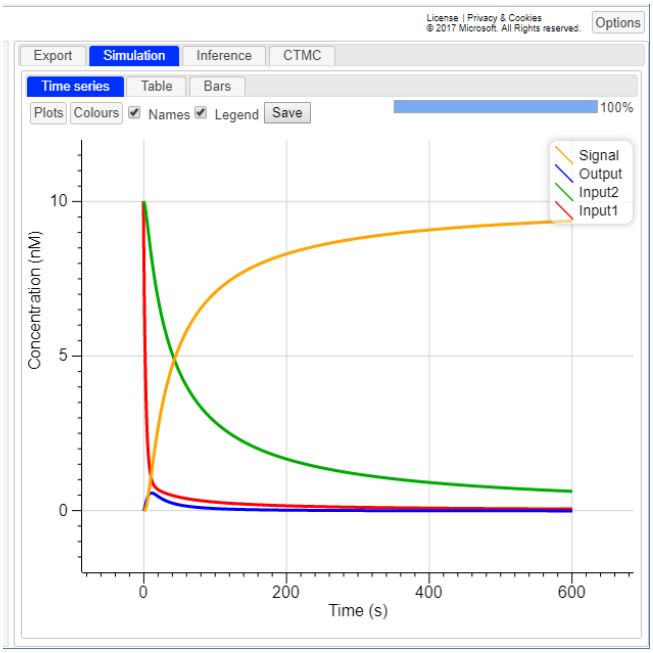
**Export/Reactions** displays the contents of the CRN/Reactions tab in a format that can be saved as a scalable vector graphics (SVG) file (Figure 5b).

**Export/SBML** displays a Systems Biology Markup Language [19] representation of the CRN/Code tab.

**Export/Matlab** displays a Matlab representation of the ordinary differential equations corresponding to the CRN/-Code tab. This export is only generated if the `simulator` directive is set to `deterministic` or `sundials`.

## 2.5 Simulation

The *Simulation* tab (Figure 6) is populated by pressing the *Simulate* button when either the DSD or CRN tabs are selected. If the DSD/Code tab is selected, pressing the Simulate button parses the DSD code and generates a




(a) Time series chart


(b) Time series table

**Figure 6: The Simulation tab.** Simulation results obtained by pressing the Simulate button for the running example in Figure 3. The results are displayed in the form of (a) a time series chart and (b) a time series table.

corresponding CRN, which is used to run the simulation and also to populate the CRN and Export tabs. Below we summarise the contents of the Simulation tab when the running example (Figure 3) is selected and the Simulate button is pressed.

### 2.5.1 Simulation/Time series

The *Time series* tab (Figure 6a) displays the results of the simulation as a chart, which is updated dynamically over time as the simulation progresses. Double-clicking on the chart resizes it to fit the tab. The chart zoom is adjusted by holding the ctrl key while scrolling with a mouse or other input device. If the pointer is over the central region of the chart then this adjusts the zoom for both the horizontal and vertical directions. If the pointer is over the horizontal axis then this adjusts the zoom in the horizontal direction only. Similarly, if the pointer is over the vertical axis then this adjusts the zoom in the vertical direction only. The chart can be saved to a file by pressing the Save button.

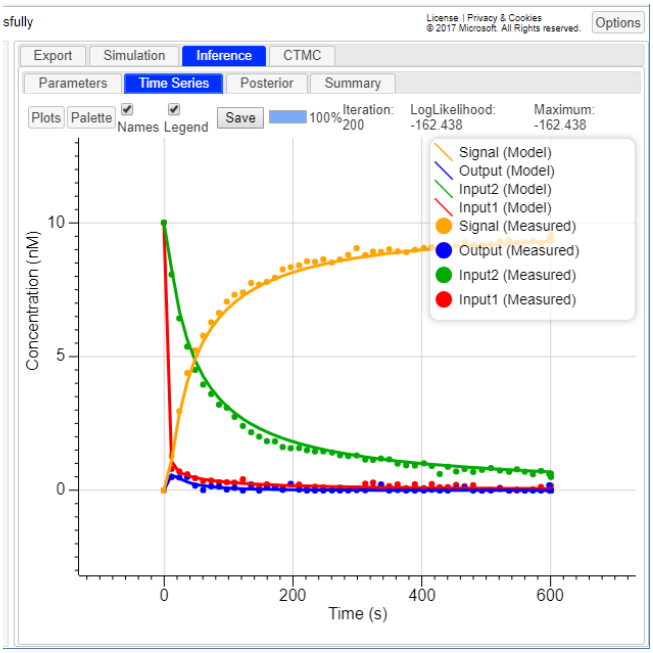
### 2.5.2 Simulation/Table

The *Table* tab (Figure 6b) displays the simulation results in table form. The results are split over multiple pages by default, to improve the user interface performance for large tables. The number of pages can be adjusted by the user. In addition, the simulation results can be saved to a file by pressing the Save button.

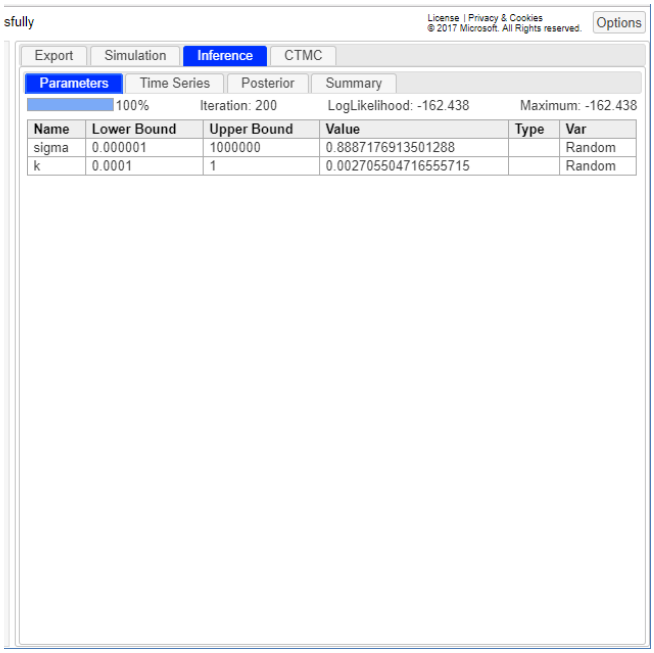
## 2.6 Inference

The *Inference* tab (Figure 7) is populated by pressing the *Infer* button when either the DSD or CRN tabs are selected. If the DSD/Code tab is selected, pressing the Infer button parses the DSD code and generates a corresponding CRN, which is used to run parameter inference and also to populate the CRN and Export tabs.

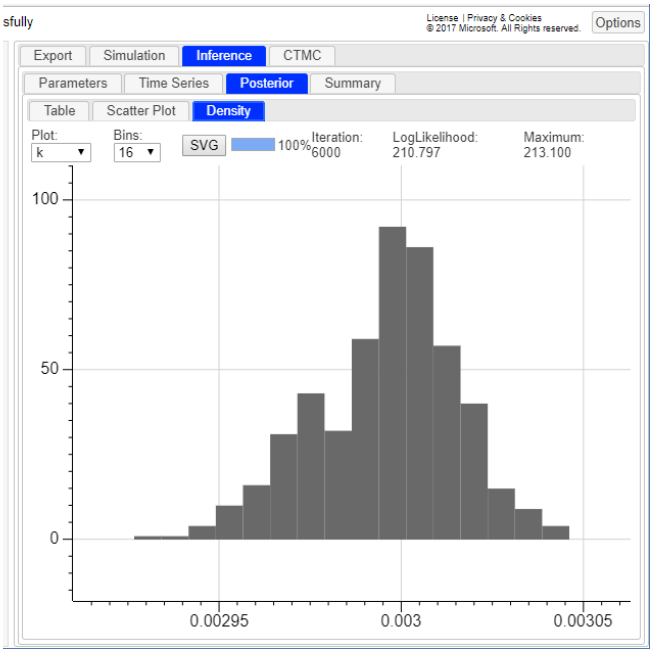
For the running example (Figure 3) there is a single parameter `k` with a fixed value of `0.003`. If corresponding experimental data is provided, the parameter can instead be allowed to vary so that it can be inferred directly from the data. This is achieved by explicitly referencing a data table from the Library/Data tab. For the running example, we add `directive data [Join_data]` to specify that the `Join_data` table in the Library/Data tab should be used. Note that this data was generated artificially for the purposes of the example. Parameters are allowed to vary by specifying a *prior* for the parameter. For the running example we replace `k = 0.003` with `k = 0.003, {interval = Log; distribution = Uniform(0.0001, 1)}`. This specifies that the parameter `k` is assumed to vary uniformly between the values of 0.0001 and 1, on a logarithmic scale. This revised example is available in the DSD/Code tab under Examples/Manual/Join - Inference. Below we summarise the contents of the Inference tab when the revised example is selected and the Infer button is pressed.



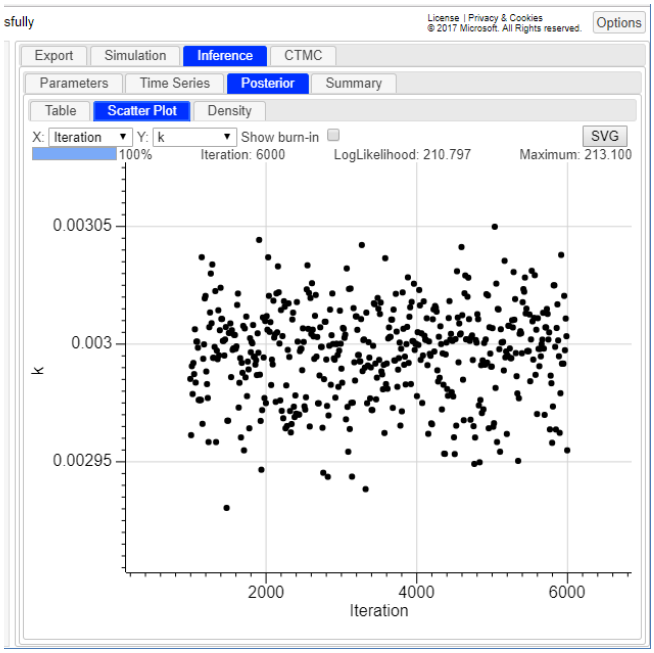

(a) Inference/Time Series tab




(b) Inference/Parameters tab




(c) Inference/Posterior/Density tab




(d) Inference/Posterior/Scatter Plot tab

**Figure 7: The Inference tab.** (a,b) Inference results obtained by pressing the Infer button for the running example in Figure 3, modified by adding `directive data [Join_data]` to specify the data to use for inference, and replacing `k = 0.003` with `k = 0.003, {interval = Log; distribution = Uniform(0.0001, 1)}` to specify the prior distribution of the parameter `k`. This revised example is available in the DSD/Code tab, under Examples/Manual/Join - Inference. (a) Time series plot of the simulation obtained using the inferred value of parameter `k`. (b) Table representation of the inferred value of parameter `k`, along with the inferred noise parameter `sigma`. (c,d) Posterior distribution of the parameter `k` obtained by running 1000 burnin and 5000 sample iterations, achieved by adding `directive inference { burnin = 1000; samples = 1000;}`. This revised example is available in the DSD/Code tab under Examples/Manual/Join - Inference - Samples. (c) Histogram of the probability density of parameter `k`. (d) Scatter plot of the values of parameter `k` for different simulation runs.

### 2.6.1 Inference/Time Series

The *Time Series* tab (Figure 7a) displays the simulation results for the different values of the parameter `k`. The results are updated for each new value of `k` that reduces the deviation between the simulation and the data.

### 2.6.2 Inference/Parameters

The *Parameters* tab (Figure 7b) displays the current value of parameter `k`. A noise parameter `sigma` is also displayed, which models the noise in the experimental data (see Section 3.7).

### 2.6.3 Inference/Posterior

The *Posterior* tab (Figure 7c,d) displays the posterior distributions of the parameters as the inference progresses. The Posterior/Density tab (Figure 7c) displays a histogram of the parameter values, and the Posterior/Scatter Plot tab (Figure 7d) displays the parameter values for individual simulation runs. More accurate parameter distributions are obtained by using a larger number of samples. This is achieved by adding `directive inference { burnin = 1000; samples = 1000;}` to the example, where `burnin = 1000` specifies that an initial 1000 simulations will be run by randomly varying the inference parameters, while `samples = 5000` specifies that a further 5000 simulations will be run and used to estimate the posterior distributions of the parameters. This produces the plots displayed in Figure 7c,d. By default, `burnin = 100` and `samples = 100`, which gives rise to much coarser distributions. Additional information about parameter inference is provided in Section 3.7.

## 2.7 CTMC

The *CTMC* tab (Figure 8) is populated by pressing the *CTMC* button when either the DSD or CRN tabs are selected. If the DSD/Code tab is selected, pressing the CTMC button parses the DSD code and generates a corresponding Continuous Time Markov Chain (CTMC). This essentially represents the reachable states of the system starting from its initial conditions, which are assumed to specify discrete populations of species. If fractions of populations are specified, these are rounded down to the nearest whole number. Each state in the CTMC corresponds to a distinct multiset of species. Transitions from a source state to a target state are determined from the reactions that are possible for the species present in the source state, and are annotated with the rate of the transition. It is important to note that the CTMC can become very large as the species populations increase. Care should be taken to ensure that the species populations are sufficiently low before the CTMC is generated. The computation of the CTMC can be stopped by pressing the Stop button. Below we briefly summarise the contents of the CTMC tab when the running example (Figure 3) is selected and the CTMC button is pressed.

### 2.7.1 CTMC/Text

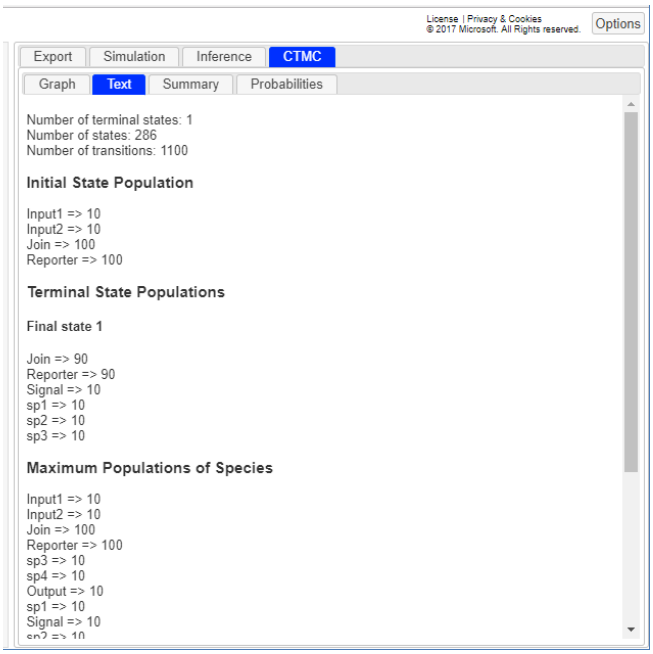
The *Text* tab (Figure 8a) summarises various properties of the CTMC, including the number of states and transitions, the initial and terminal states, and the upper and lower bounds of the species. For the running example, the CTMC has 286 states and 1100 transitions. One of the states is *terminal*, meaning that it has no outgoing transitions. The initial state contains 10 Input1, 10 Input2, 100 Join and 100 Reporter species, while the terminal state contains 90 Join, 90 Reporter and 10 Signal species, together with 10 sp1, sp2 and sp3 species. This terminal state is reached by consuming all of the Input1 and Input2 species, together with 10 of the Join and Reporter species, to produce 10 Signal species.

### 2.7.2 CTMC/Summary

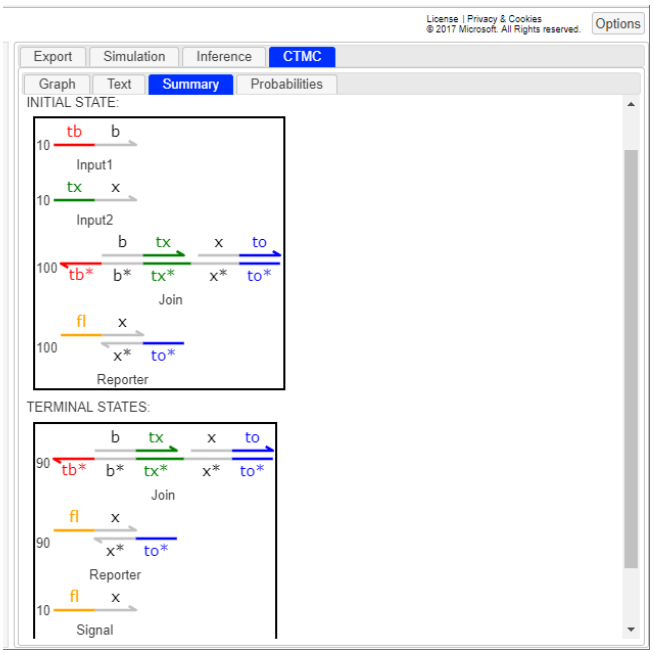
The *Summary* tab (Figure 8b) displays a graphical representation of the initial and terminal states. Note that the species are consistent with those displayed in the CRN, DSD and Export tabs.

### 2.7.3 CTMC/Graph

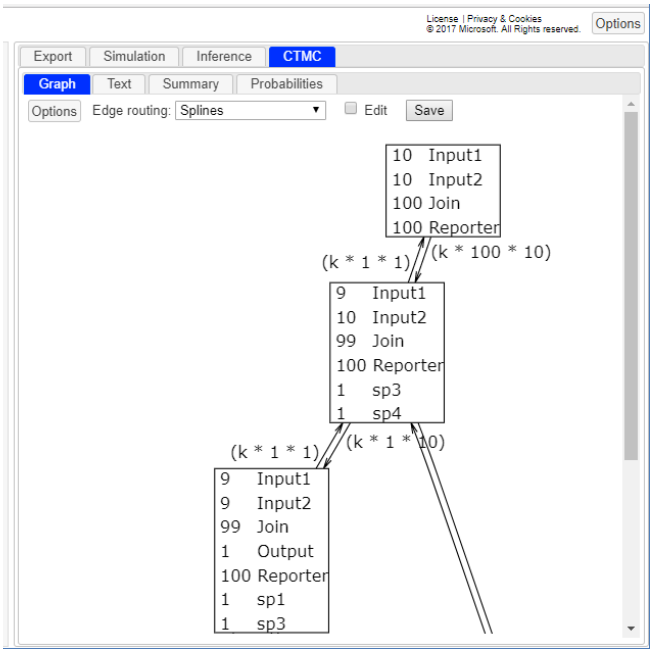
The *Graph* tab (Figure 8c) displays the full CTMC as a graph, where each node in the graph represents a reachable state, and each edge represents a transition between two states. For the running example, the visualisation takes several seconds to render, due to the relatively large numbers of states and transitions. The large size of the CTMC stems from the fact that the different interactions between species can happen in different orders, starting from the given initial state. Here we have zoomed in on the top portion of the CTMC by using the mouse scroll wheel, to focus on the initial transitions. We observe that the first transition consumes one Input1 and one Join species, and produces one sp3 and one sp4 species. The transition happens with rate k*10*100 since there are 10 Input1 species and 100 Join species that can interact, with each individual interaction occurring at rate k. The transition is reversible, since sp3 and sp4 can interact to produce the original Input1 and Join species. We note that the second state in the figure has two other possible transitions. One of these transitions involves Input2 interacting with sp4 to produce Output and sp1. The other transition is only partially displayed, but involves one of the 9 remaining Input1 species interacting with one of the 99 remaining Join species. Note that all of these transitions are consistent with the reactions present in the CRN (Figure 4), with the rate of a single reaction scaled by the number of ways in which that reaction can take place.



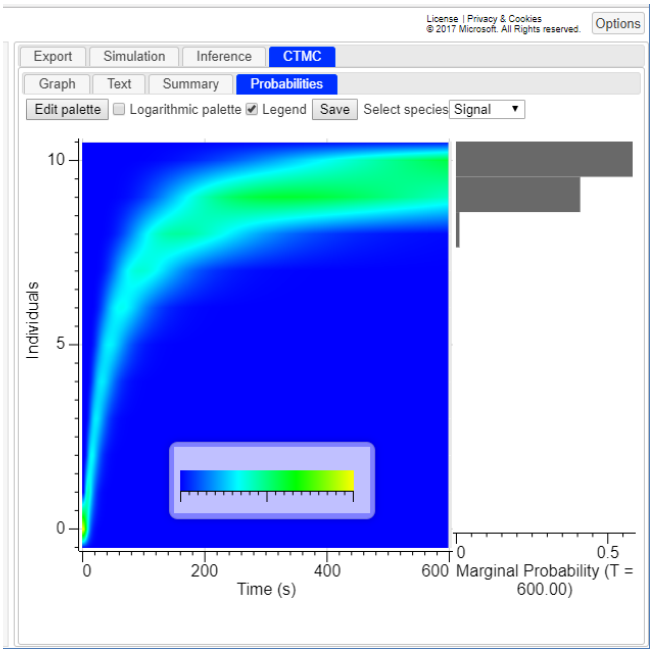

(a) CTMC/Text tab




(b) CTMC/Summary tab




(c) CTMC/Graph tab




(d) CTMC/Probabilities tab

**Figure 8: The CTMC tab.** (a-c) Results obtained by pressing the CTMC button for the running example in Figure 3. (a) A summary of various properties of the CTMC. (b) The initial and terminal states of the CTMC. (c) A zoomed in portion of the CTMC, starting from the initial state. (d) Probability map of the Signal species over time, obtained by replacing `directive simulator deterministic` with `directive simulator cme` in the running example and pressing the Simulate button. This revised example is available in the DSD/Code tab, under Examples/Manual/Join - CTMC.

### 2.7.4 CTMC/Probabilities

The *Probabilities* tab (Figure 8c) requires a Chemical Master Equation (CME) simulation to be run. This is achieved by replacing `directive simulator deterministic` with `directive simulator cme` and pressing the Simulate button. Additional details about this simulation method are provided in Section 3.5.4. Running the CME simulator generates the CTMC and at the same time computes the probabilities of a given species having a given value at a given time. The probabilities are shown as a heat map, which is always scaled between 0 and 1 for a given time point. For the running example, the probabilities of the Signal species are shown, and different species can be selected via a drop-down menu. A histogram of the probability distribution for a given time point is shown on the

right hand side of the heat map. The histogram shows the probability of the chosen species having a given discrete value at the chosen time point. The final time point is selected by default, and earlier time points can be selected by clicking on the heat map. For the running example, we observe that the `Signal` species has a high probability of having a population of either 9 or 10 at the final time point, and a much lower probability of having a population of 8. The probability of the `Signal` species having a population less than 8 at this time point is effectively zero. However, we can see from the heat map that this probability is much higher near the beginning of the simulation, since this species is initialised with a population of 0.

# 3 Visual CRN Language

This section provides a detailed description of the Visual CRN language. Most of the language features are also shared with the Visual DSD language. We use the Visual CRN program generated in Figure 4A as the basis for a running example.

## 3.1 Syntax Conventions

We first summarise the main syntax conventions of the Visual CRN language. Terminal symbols are written in `teletype` font and non-terminal symbols are written in *italics*. Comments begin with `(*` and end with `*)` and may be nested. Single line comments are written using `//` at the start of the comment. We use a standard syntax for *Records*, *Lists* and *Maps*, and assume that $N \geq 0$ unless stated otherwise:

**Records** are written as $\{Name_1 = Field_1; \ldots; Name_N = Field_N\}$, where $N \geq 1$. We assume that each *Name* in the record is unique and that each *Field* can potentially have a different type. We allow a *Field* to be omitted, in which case a default value is used.

**Lists** are written as $List\langle Value\rangle$, which is short for the syntax $[Value_1; \ldots; Value_N]$. We assume that every *Value* is of the same type.

**Maps** are written as $Map\langle Key, Value\rangle$, which is short for the syntax $[Key_1 = Value_1; \ldots; Key_N = Value_N]$. We assume that each *Key* in the map is unique and that every *Value* is of the same type.

We use the following syntax for *Integers*, *Names*, *Strings*, *Floats* and *Keywords*:

**Integers** are non-empty sequences of digits, where a digit denotes a single character in the range 0-9,

**Names** are sequences of characters, where the first character must either be a letter, which is a character in the range A-Z or a-z, or the underscore character (_). This is followed by a possibly-empty sequence of characters which may be letters, digits, underscores or apostrophes (').

**Strings** are possibly-empty sequences of characters enclosed by quotation marks (`"`). Any quotation marks appearing within the string must be escaped by a preceding backslash (\).

**Floats** can be written in three different ways: (i) One or more digits followed by a decimal point (.), followed by zero or more digits. For example 3.141. (ii) One or more digits followed by an uppercase E or lowercase e, followed by a plus (+) or minus (-) sign, followed by one or more digits. For example 3e-5. (iii) One or more digits followed by a decimal point, followed by zero or more digits, followed by an uppercase E or lowercase e, followed by a plus or minus sign, followed by one or more digits. For example 1.4324e+2.

**Keywords** are reserved words that cannot be used as a *Name*. The list of keywords is as follows: `constant directive def dom else false float_of_int if init int_of_float log new not prod rxn sum tether then time true`

## 3.2 CRN Programs

A *Program* in the Visual CRN language consists of multiple *Settings*, followed by multiple *Modules*, followed by a *Process*. The *Settings* are parameterised by the type of *Species* and are defined in Section 3.3. For the Visual CRN language the *Species* is simply a *Name*, however for the Visual DSD language it is a more complex structure made up of DNA strands. A *Module* associates a module *Name* and multiple arguments $(Name_1, \ldots, Name_N)$ with a *Process*, which can be an *Initial* condition, a *Reaction*, a module *Name* instantiated with multiple values, or multiple parallel processes, separated by the parallel composition operator (`|`). Example CRN programs are described in Figure 9.

| | |
|---|---|
| $Program ::= Settings\langle Species\rangle\ Modules\ Process$ | CRN Program |
| $Settings\langle S\rangle ::= \texttt{directive}\ Setting\langle S\rangle_1 \ldots \texttt{directive}\ Setting\langle S\rangle_N$ | Multiple settings |
| $Modules ::= \texttt{module}\ Module_1 \ldots \texttt{module}\ Module_N$ | Multiple modules |
| $Module ::= Name(Name_1, \ldots, Name_N) = \{Process\}$ | Module definition |
| $Species ::= Name$ | CRN species |
| $Process ::=$ | |
| $\mid Initial\langle Species\rangle$ | Initial condition |
| $\mid Reaction\langle Species\rangle$ | Reaction |
| $\mid Name(Value_1, \ldots, Value_N)$ | Module instance |
| $\mid Process_1 \mid \ldots \mid Process_N$ | Multiple parallel processes |

**A**

```
1  directive simulation {
2    final=600;
3    plots=[Input1; Input2; Output; Signal];
4  }
5  directive simulator deterministic
6  directive parameters [ k = 0.003; u = 0.1 ]
7  | 100 Reporter
8  | 100 Join
9  | 10 Input2
10 | 10 Input1
11 | Reporter + Output ->{k} sp2 + Signal
12 | Join + Input1 <->{k}{k} sp4 + sp3
13 | sp4 + Input2 <->{k}{k} sp1 + Output
```

**B**

```
1  directive simulation {
2    final=600;
3    plots=[Input1; Input2; Output; Signal];
4  }
5  directive simulator deterministic
6  directive parameters [ k = 0.003; u = 0.1 ]
7  module Reversible(A,B,f,r,C,D) = {
8    | A + B <->{forward}{reverse} C + D
9  }
10 | 100 Reporter
11 | 100 Join
12 | 10 Input2
13 | 10 Input1
14 | Reporter + Output ->{k} sp2 + Signal
15 | Reversible(Join,Input1,k,k,sp4,sp3)
16 | Reversible(sp4,Input2,k,k,sp1,Output)
```

**Figure 9: Example CRN programs. A** CRN program from Figure 4A, which was automatically generated from the DSD program in Figure 3A. Lines 1-6 contain CRN *Settings*, while lines 7-13 contain a *Process*, consisting of four *Initial* conditions (lines 7-10), and three *Reactions* (lines 11-13). **B** CRN program with modules that is equivalent to the program in (A). The `Reversible` module takes arguments `A,B,f,r,C,D` and creates a reaction with reactants `A+B`, products `C+D`, forward rate `f` and reverse rate `r` . The two instances of this module on lines 15 and 16 are equivalent to the reactions on lines 12 and 13 of program (A), respectively, since the arguments of the instances replace the arguments of the module.

**A**

```
1  directive simulation {
2    final=600;
3    plots=[Input1; Input2; Output; Signal];
4  }
5  directive simulator deterministic
6  directive parameters [ k = 0.003; u = 0.1 ]
7  | constant 100 Reporter
8  | constant 100 Join
9  | 10 Input2 @ 50
10 | 10 Input1 @ 50
11 | Reporter + Output ->{k} sp2 + Signal
12 | Join + Input1 <->{k}{k} sp4 + sp3
13 | sp4 + Input2 <->{k}{k} sp1 + Output
```

**B**

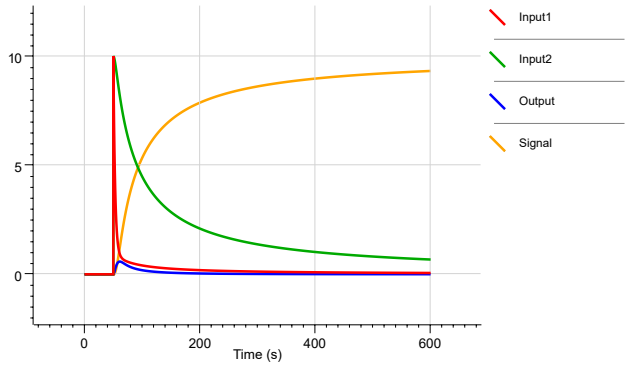



**Figure 10: Example initial conditions. A** CRN program from Figure 9, updated with constant and delayed initial conditions. The `Reporter` and `Join` species are assumed to be in excess, however they will be consumed over time if they are not fixed as `constant`, slowing down the release of the `Signal` strand. Fixing these species as `constant` is a convenient way to study the effect of species loss over time on system kinetics. In additiona, in a laboratory experiment the `Reporter` and `Join` species will typically be mixed in solution beforehand, prior to adding the `Input1` and `Input2` species a short time later. This can be modelled by delaying the addition of the two input species. In this example both species are added after 50 time units. **B** Simulation results for the CRN program in (A). We observe a discrete spike in the concentrations of both inputs after 50 time units. The `Reporter` and `Join` species remain constant at 100 throughout the duration of the simulation (not shown), however this does not seem to significantly alter the system dynamics, suggesting that species loss does not play a key role in this particular simulation.

### 3.2.1 Initial Conditions

A species can be initialised with a value, where*Value S* initialises species *S* with the given *Value*. The initialisation can also be delayed, where $Value_1$ *S* @$Value_2$ initialises species *S* with $Value_1$ after time $Value_2$ . In addition, a species can be fixed as constant, where `constant` *Value S* ensures that species *S* remains constant at the given *Value* throughout the duration of the simulation. A more general syntax can also be used to initialise a species, where $S = Attributes$ initialises species *S* with the specified *Attributes*. For example, $Value_1$ *S* @$Value_2$ can be written as $S = \{\texttt{value} = Value_1; \texttt{time} = Value_2\}$, and `constant` *Value S* can be written as $S = \{\texttt{value} = Value; \texttt{constant} = \texttt{true}\}$. Additional attributes can also be used for spatial placement of species, defined in Section 3.6.

| | |
|---|---|
| $Initial\langle S\rangle ::=$ | |
| \| *Value S* | Initial species population |
| \| $Value_1$ *S* @$Value_2$ | Delayed species population |

| | |
|---|---|
| \| `constant` *Value S* | Constant species population |
| \| *S* = *Attributes* | Species with attributes |

| | |
|---|---|
| *Attributes* ::= { | Attributes record |
| `value` = *Value*; | Species value (1) |
| `time` = *Value*; | Species delay (0) |
| `constant` = *Boolean*; | Species is constant (`false`) |
| `spatial` = *Placement*; | Species placement |
| } | |

### 3.2.2 Reactions

The syntax of reactions is defined as follows:

| | |
|---|---|
| *Reaction*⟨*S*⟩ ::= | |
| \| *Reactants*⟨*S*⟩ `->`{*Value*} *Multiset*⟨*S*⟩ | Mass-action reaction |
| \| *Reactants*⟨*S*⟩ `<->`{*Value*}{*Value*} *Multiset*⟨*S*⟩ | Reversible mass-action reaction |
| \| *Reactants*⟨*S*⟩ `->`[*Functional*⟨*S*⟩] *Multiset*⟨*S*⟩ | Functional rate reaction |
| \| *Reactants*⟨*S*⟩ `<->`[*Functional*⟨*S*⟩][*Functional*⟨*S*⟩] *Multiset*⟨*S*⟩ | Reversible functional rate reaction |

| | |
|---|---|
| *Reactants*⟨*S*⟩ ::= | |
| \| *Multiset*⟨*S*⟩ | Multiset of reactants |
| \| *Multiset*⟨*S*⟩∼*Multiset*⟨*S*⟩ | Multisets of catalysts and reactants |

| | |
|---|---|
| *Multiset*⟨*S*⟩ ::= $Value_1\ S_1 + \cdots + Value_N\ S_N$ | Multiset of species |

### 3.2.3 Expressions

The syntax of expressions is defined as follows:

| | |
|---|---|
| *Value* ::= *Expression*⟨*Name*⟩ | Expression over parameters |
| *Functional*⟨*S*⟩ ::= *Expression*⟨*Key*⟨*S*⟩⟩ | Expression over species and parameters |

| | |
|---|---|
| *Key*⟨*S*⟩ ::= | |
| \| [*S*] | Species |
| \| [`time`] | Simulation time |
| \| [*Name*] | Functional rate |
| \| *Name* | Parameter name |

| | |
|---|---|
| *Expression*⟨*T*⟩ ::= | |
| \| *T* | Expression variable |
| \| *Float* | Floating point number |
| \| *Expression*⟨*T*⟩ + *Expression*⟨*T*⟩ | Addition |
| \| *Expression*⟨*T*⟩ − *Expression*⟨*T*⟩ | Subtraction |
| \| *Expression*⟨*T*⟩ ∗ *Expression*⟨*T*⟩ | Multiplication |
| \| *Expression*⟨*T*⟩ / *Expression*⟨*T*⟩ | Division |
| \| *Expression*⟨*T*⟩%*Expression*⟨*T*⟩ | Modulo |
| \| *Expression*⟨*T*⟩^*Expression*⟨*T*⟩ | Power |
| \| *Expression*⟨*T*⟩ ∗ ∗*Expression*⟨*T*⟩ | Power |
| \| `sum`(*List*⟨*Expression*⟨*T*⟩⟩) | Sum |
| \| `prod`(*List*⟨*Expression*⟨*T*⟩⟩) | Product |
| \| `log`(*Expression*⟨*T*⟩) | Logarithm |
| \| `if` *Condition*⟨*T*⟩ | Conditional |
| `then` *Expression*⟨*T*⟩ | |
| `else` *Expression*⟨*T*⟩ | |

| | |
|---|---|
| *Condition*⟨*T*⟩ ::= | |
| \| `true` | Logical True |
| \| `false` | Logical False |
| \| *Expression*⟨*T*⟩ <= *Expression*⟨*T*⟩ | Less than or equal |
| \| *Expression*⟨*T*⟩ < *Expression*⟨*T*⟩ | Less than |

| *Expression*⟨*T*⟩ = *Expression*⟨*T*⟩ | Equal |
|---|---|
| *Expression*⟨*T*⟩ > *Expression*⟨*T*⟩ | Greater than |
| *Expression*⟨*T*⟩ >= *Expression*⟨*T*⟩ | Greater than or equal |
| *Condition*⟨*T*⟩&&*Condition*⟨*T*⟩ | Logical AND |
| *Condition*⟨*T*⟩\|\|*Condition*⟨*T*⟩ | Logical OR |
| `not` *Condition*⟨*T*⟩ | Negation |

## 3.3 CRN Settings

Settings are defined in terms of records, maps and lists:

*Setting*⟨*S*⟩ ::=

| | |
|---|---|
| `parameters` *Map*⟨*Name*, *Parameter*⟩ | Parameter definitions |
| `sweeps` *Map*⟨*Name*, *List*⟨*Assignment*⟩⟩ | Sweep definitions |
| `rates` *Map*⟨*Name*, *Functional*⟨*S*⟩⟩ | Rate expressions |
| `plot_settings` *Plotting* | Plot format settings |
| `simulation` *Simulation*⟨*S*⟩ | Simulation settings |
| `simulations` *Map*⟨*Name*, *Simulation*⟨*S*⟩⟩ | Settings for multiple simulations |
| `simulator` *Simulator* | Simulator method (`stochastic`) |
| `deterministic` *Deterministic* | Deterministic simulation settings |
| `stochastic` *Stochastic* | Stochastic simulation settings |
| `spatial` *Spatial*⟨*S*⟩ | Spatial simulation settings |
| `moments` *Moments*⟨*S*⟩ | Moment closure simulation settings |
| `inference` *Inference* | Inference settings |
| `data` *List*⟨*Name*⟩ | Data files for inference |
| `units` *Units* | Units settings |

### 3.3.1 Parameters

Parameters are assigned a value and an optional prior:

*Parameter* ::=

| | |
|---|---|
| *Value* | Parameter value |
| *Value*, *Prior* | Parameter value and associated prior |

### 3.3.2 Sweeps

Sweeps are used to assign values to parameters:

*Assignment* ::=

| | |
|---|---|
| *Name* = *Value* | Assign a value to a parameter name |
| $(Name_1, \ldots, Name_N) = (Value_1, \ldots, Value_N)$ | Assign a set of values to a set of names |

### 3.3.3 Units

Units for time, space and concentration are used in a consistent manner for rate constants and simulation plots. These units can be modified using the following settings:

| | |
|---|---|
| *Units* ::= { | Units record |
| `time` = *Time*; | Time units (`s`) |
| `space` = *Space*; | Space units (`m`) |
| `concentration` = *Concentration*; | Concentration units (`nM`) |
| } | |

*Time* ::=

| | |
|---|---|
| `h` | hours (3600s) |
| `min` | minutes (60s) |
| `s` | seconds (s) |
| `ms` | milliseconds ($10^{-3}$ s) |
| `us` | microseconds ($10^{-6}$ s) |
| `ns` | nanoseconds ($10^{-9}$ s) |

*Space* ::=

| | |
|---|---|
| `m` | metres (m) |
| `mm` | millimetres ($10^{-3}$ m) |
| `um` | micrometres ($10^{-6}$ m) |
| `nm` | nanometres ($10^{-9}$ m) |
| `pm` | picometres ($10^{-12}$ m) |
| `fm` | femtometres ($10^{-15}$ m) |

*Concentration* ::=

| | |
|---|---|
| `M` | molar (M) |
| `mM` | millimolar ($10^{-3}$ M) |
| `uM` | micromolar ($10^{-6}$ M) |
| `nM` | nanomolar ($10^{-9}$ M) |
| `pM` | picomolar ($10^{-12}$ M) |
| `fM` | femtomolar ($10^{-15}$ M) |
| `aM` | attomolar ($10^{-18}$ M) |
| `zM` | zeptomolar ($10^{-21}$ M) |
| `yM` | yoctomolar ($10^{-24}$ M) |

### 3.3.4 Simulation

The `simulation` directive specifies the simulation settings to be used when the Simulate button is pressed. The *Simulation* settings are defined below, with default values shown in parentheses.

| | |
|---|---|
| *Simulation*⟨*S*⟩ ::= { | Simulation record |
| `initial` = *Float*; | Initial simulation time (0) |
| `final` = *Float*; | Final simulation time (1000) |
| `points` = *Integer*; | Number of simulation points to plot (1000) |
| `plots` = *Map*⟨*Plot*⟨*S*⟩⟩; | Species to plot. Plots all species if empty ([]) |
| `kinetics` = *Kinetics*; | Kinetic rate convention for homomultimer formation (`Contextual`) |
| `prune` = *Boolean*; | Remove unreachable reactions prior to simulation (`false`) |
| `multicore` = *Boolean*; | Use multiple CPU cores to run simulation, if present (`false`) |
| `data` = *List*⟨*Name*⟩; | List of Datasets to compare with simulation results ([]) |
| `sweeps` = *List*⟨*Name*⟩; | List of sweeps to use for simulation ([]) |
| } | |

| | |
|---|---|
| *Plot*⟨*S*⟩ ::= | |
| \| *S* | Single species |
| \| *Functional*⟨*S*⟩ | Function of species, parameters or rates |

| | |
|---|---|
| *Kinetics* ::= | |
| \| `Contextual` | Kinetic convention depends on simulator directive |
| \| `Stochastic` | Stochastic kinetic convention |
| \| `Deterministic` | Deterministic kinetic convention |

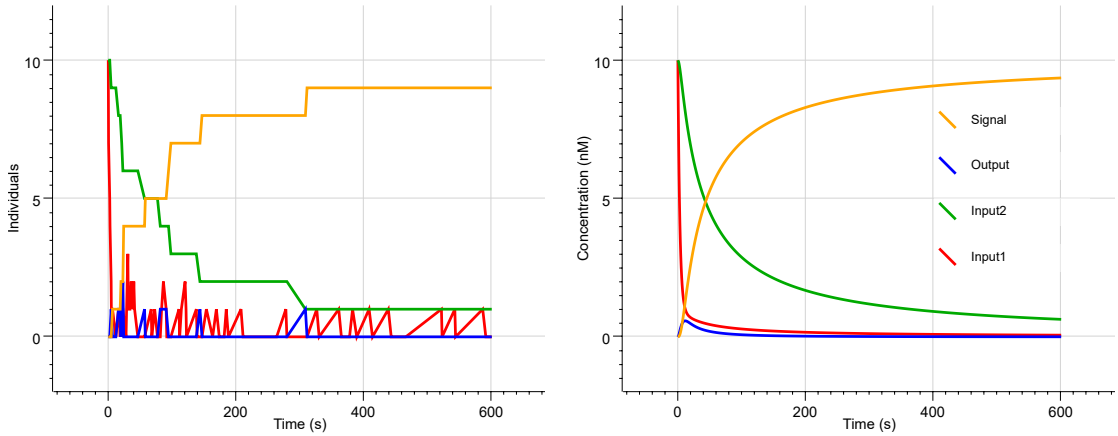
The `kinetics` setting defines the kinetic rate law to be used for simulation. This is needed due to an important difference between the kinetic rate laws of stochastic and deterministic systems, specifically in the case of *homomultimer* formation, which is the formation of complexes involving multiple copies of the same molecular species. The stochastic rate law includes a $k!$ divisor, where $k$ is the multiplicity of the species, which is not present in the deterministic rate law. For example, the reaction $A + A \rightarrow^k B$ has a deterministic rate law of $k[A]^2$, where $[A]$ represents the *concentration* of species $A$, but a stochastic rate law of $k[A]([A]-1)/2$, where $[A]$ represents the *population* of species $A$. This difference means that the limiting behaviour of the stochastic mean with increasing copy numbers does not converge to the deterministic rate equations, as is commonly thought. Only when the stochastic rate laws are equal to the deterministic rate laws will this be the case. By default, `kinetics` is set to `Contextual`, which means that a deterministic rate law is assumed for the `deterministic` simulator, while a stochastic rate law is assumed for the `stochastic`, `cme`, `lna` and `mc` simulators. Alternatively, the kinetic law can be fixed to either the `Stochastic` or `Deterministic` convention, so that it is the same for all simulators.

### 3.3.5 Simulator

The `simulator` directive specifies the simulator to be used when the Simulate button is pressed. The *Simulator* settings are defined below. The default simulator is `stochastic`.

| | |
|---|---|
| *Simulator* ::= | |
| \| `deterministic` | Deterministic simulation of ordinary differential equations (OSLO solvers) |
| \| `sundials` | Deterministic simulation of ordinary differential equations (SUNDIALs solvers) |
| \| `stochastic` | Stochastic simulation using the Gillespie Stochastic Simulation Algorithm (SSA) |
| \| `lna` | Mean and variance over time of a stochastic system (Linear Noise Approximation) |
| \| `cme` | Probability distribution over time of a stochastic system (Chemical Master Equation) |
| \| `pde` | Deterministic spatiotemporal simulation of reaction-diffusion equations |

Deterministic simulation can be performed using two different solvers for numerical integration. Selecting the `deterministic` simulator uses the Oslo Solving Library for ODEs (OSLO) solver. When using the downloadable version it is possible to use the SUNDIALs integrators [20] by selecting the `sundials` simulator. SUNDIALs is a C-based library, and consequently simulations run faster than when using OSLO.




(a) Stochastic simulation (b) Deterministic simulation

**Figure 11: Comparison of `stochastic` and `deterministic` simulation.**

## 3.4 Deterministic Simulation

### 3.4.1 Deterministic Settings

The *Deterministic* settings are defined below:

| | |
|---|---|
| *Deterministic* ::= { | Deterministic record |
| `stiff` = *Boolean*; | Enable stiff solver (`false`) |
| `abstolerance` = *Float*; | Relative tolerance (1E-5) |
| `reltolerance` = *Float*; | Absolute tolerance (1E-6) |
| } | |

We note that the stiff solver is more efficient for *stiff* problems, which are those that incorporate different timescales in their dynamics. The `deterministic` solver uses Gear's method when `stiff = true` and Runge-Kutta 5(4) when `stiff=false`. The stiff solver is disabled by default, since it is more sensitive to numerical error than the non-stiff solver. For more information on tolerances, see the documentation for OSLO or SUNDIALs directly.

### 3.4.2 Deterministic Simulation Method

By specifying `directive simulator deterministic`, a CRN system is converted to a system of ordinary differential equations (ODEs) describing its *rate equations*. For each reaction $i$, a *velocity* $v_i$ is computed. For mass action reactions, the velocities are defined by

$$v_i = k_i \prod_j [c_j]^{r_{ij}} \tag{1}$$

where $[c_i]$ is the concentration of species $i$, $r_{ij}$ is the multiplicity of the $j^{\text{th}}$ species in reaction $i$, and $k_j$ is the rate constant. Note that the matrix $R := (r_{ij})$ is known as the *reaction stoichiometry matrix*. For functional reactions, the velocity function is simply taken as the function specified in the reaction syntax.

By storing the stoichiometry $S \in \mathbb{R}^{n_S, n_R}$ of the species updates of each reaction (i.e. how many copies are gained/lost from the reaction firing), the rate equations can be defined as

$$\frac{d\mathbf{c}}{dt} = S.\mathbf{v} \tag{2}$$

where $\mathbf{c}$ is the vector containing the concentrations of each molecule $c_i$.

## 3.5 Stochastic Simulation

### 3.5.1 Stochastic Settings

The *Stochastic* settings are defined below:

| | |
|---|---|
| *Stochastic* ::= { | Stochastic record |
| `scale` = *Float*; | Scale volume while fixing concentrations (1) |

`seed` = *Option*⟨*Integer*⟩; Random number generator seed (∅, uses random seed)
`steps` = *Option*⟨*Integer*⟩; Number of simulation steps to take (∅, does not limit steps)
`trajectories` = *Integer*; Number of simulation repeats, plotting mean and variance (1)
}

In order to perform a stochastic simulation, concentrations must be converted to numbers of individuals. This can be achieved using the following equation:

$$n = \lceil c.V.N_A \rceil \tag{3}$$

where $n$ is the number of individuals, $c$ is the concentration, $V$ is the volume and $N_A$ is Avogadro's constant, which denotes the number of individuals per mole of substance (approximately $6.02214 \times 10^{23}$ molecules). The function $\lceil x \rceil$ denotes the rounding up of $x$ to its nearest natural number. Thus, in order to convert a concentration into a number of individuals, it is sufficient to multiply the concentration by a scale factor $s = V.N_A$, which denotes the number of individuals per unit concentration. Essentially, this corresponds to choosing a volume $V$ such that the number of individuals is equal to $s$ for one unit of concentration. For example, a scale factor of 50 corresponds to a volume that is 50 times the volume occupied by a single individual. The units of the scale factor are therefore assumed to be the inverse of the unit of concentration. Note that the conversion from concentrations to individuals is achieved using a scale factor $s$ rather than specifying a volume $V$ directly, since it is difficult to choose a volume such that the number of individuals is a natural number. The scale factor can be set by the scale directive, where the default scale factor is 1.0.

Note that the units for rate constants are assumed to be consistent with the units for time and concentration. For example, if the unit for time are s and the unit for concentration are nM, then the unit for the bimolecular rate constants are assumed to be $\text{nM}^{-1}\ \text{s}^{-1}$, and the unit for the unimolecular rate constants are assumed to be $\text{s}^{-1}$. Once a suitable scale factor has been selected, in order to perform a stochastic simulation the molar concentrations are multiplied by the scale factor, while the concentration-dependent rates are divided by the scale factor. For example, if the scale factor is 100 $\text{nM}^{-1}$ then a concentration-dependent rate of 0.4 $\text{nM}^{-1}\ \text{s}^{-1}$ is converted to a stochastic rate of 0.004 $\text{s}^{-1}$ for simulation. Additional details on converting between populations and concentrations can be found in Section 4.2 of [21], including specific conversion rules for homodimerization reactions.

### 3.5.2 Stochastic Simulation Algorithm (SSA)

The `stochastic` simulator uses Gillespie's stochastic simulation algorithm (SSA) to generate a possible trajectory of the system over time. The stochastic dynamics is equivalent to the deterministic ODEs in the limit where the population counts approach infinity. The difference between the plots can be seen by comparing the plots produced for the running example by the stochastic (left) and deterministic (right) simulators below.

### 3.5.3 Linear Noise Approximation (LNA)

The linear noise approximation (LNA) is the simplest approximation of a stochastic simulation that is supported. The LNA approximates the mean and standard deviation of the copy number distribution for each species, under the assumption that the distribution is Gaussian. Therefore, this is not appropriate when the distribution is expected to be bimodal, or strongly skewed. Since only the mean and standard deviation are computed, the marginal distribution plot (Probability tab) is not used. However, by default, the mean is plotted with a single standard deviation area plot (Figure 12).

To evaluate an approximate mean and variance of a CRN, we can use the linear noise approximation (LNA), which evaluates the first and second order statistics in the limit as the copy number goes to infinity. The solution can be obtained numerically, by integrating the following system of equations

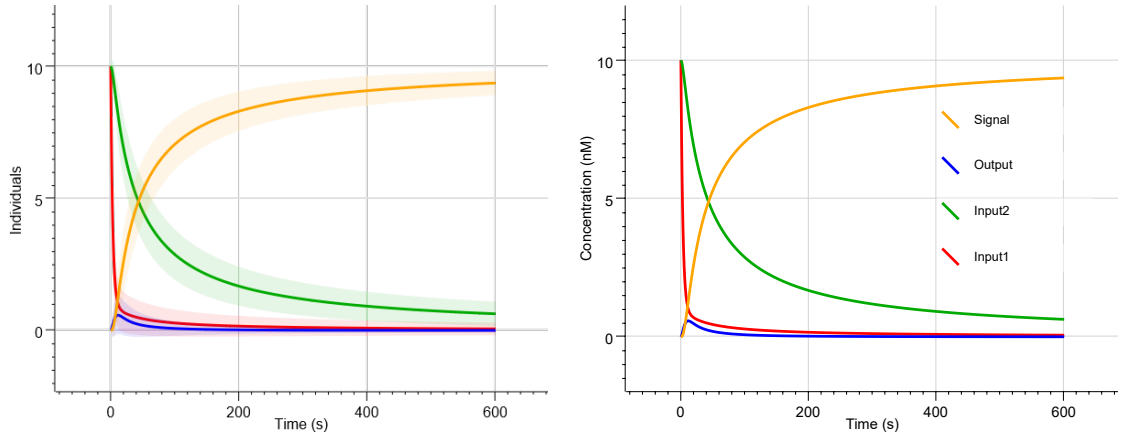
$$d \tag{4}$$

Here, $J$ is the Jacobian matrix of first-order partial derivatives of the ODEs $f$, with respect to each species. This is given by

$$J = \frac{\partial}{\partial x}\left(S.v(x)\right) = S.\frac{\partial v(x)}{\partial x} \tag{5}$$

The $(i,j)^{\text{th}}$ entry of $\dfrac{\partial v(x)}{\partial x}$ is given by

$$\frac{\partial v_i}{\partial x_j} = k_i.r_{ij}.x_j^{r_{ij}-1}.\prod_{k \neq j}[x_k]^{r_{ik}} \tag{6}$$




(a) LNA simulation

(b) Deterministic simulation

**Figure 12: Comparison of `lna` and `deterministic` simulation.**

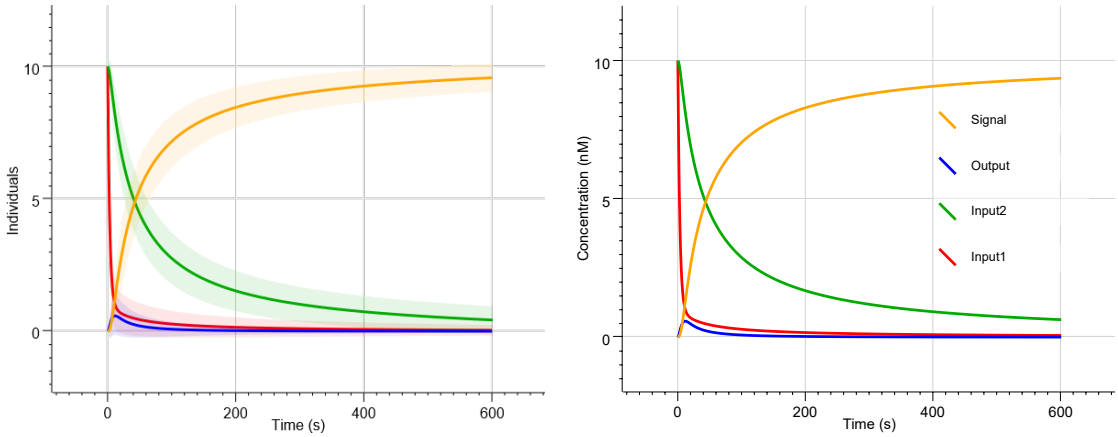
### 3.5.4 Chemical Master Equation (CME)

Stochastic simulations can be thought of as being random samples from a probability distribution. The distribution of molecule copy numbers can be described by the *chemical master equation* (CME), which can be evaluated directly. By selecting the `cme` simulator, the CME is integrated numerically, producing the probability of each *state* (specific molecule numbers of each species) over time. The simulator returns the mean ($\mu_S$) and standard deviation ($\sigma_S$) of the marginal distribution of each species $S$, plotted as a solid line for $\mu_S$ and an area plot containing the region bounded by $\mu \pm \sigma_S$ (Figure 13a). Additionally, the `cme` simulator enables the full marginal distributions to be plotted (Figure 13c-f ).

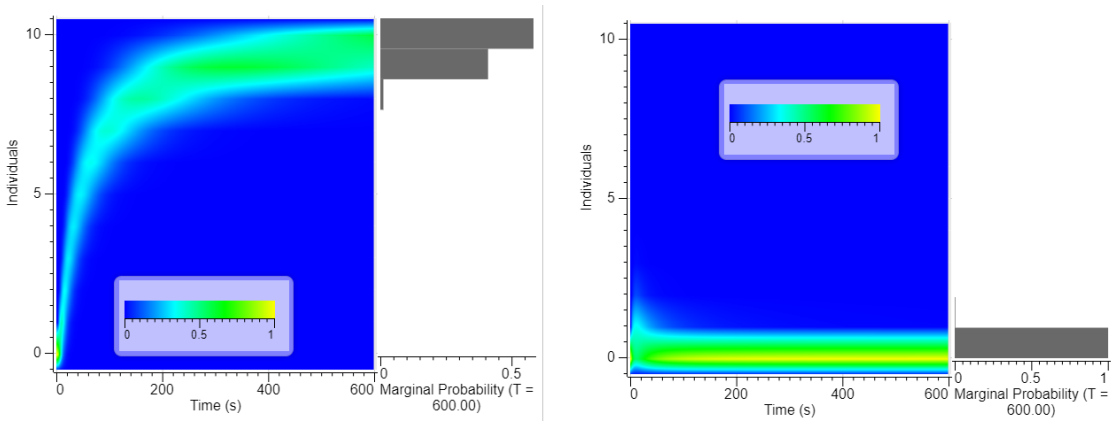
The fundamental limitation of the CME is that all discrete states of the system must be enumerated. The number of states grows very quickly, and is sometimes unbounded, for instance when there are reactions that create mass, such as $\emptyset \rightarrow X$. When unbounded, state space enumeration will not terminate, and therefore the software will crash when it runs out of memory. When bounded, but large, the simulator will take a long time.

### 3.5.5 Comparison of Stochastic Simulation Methods

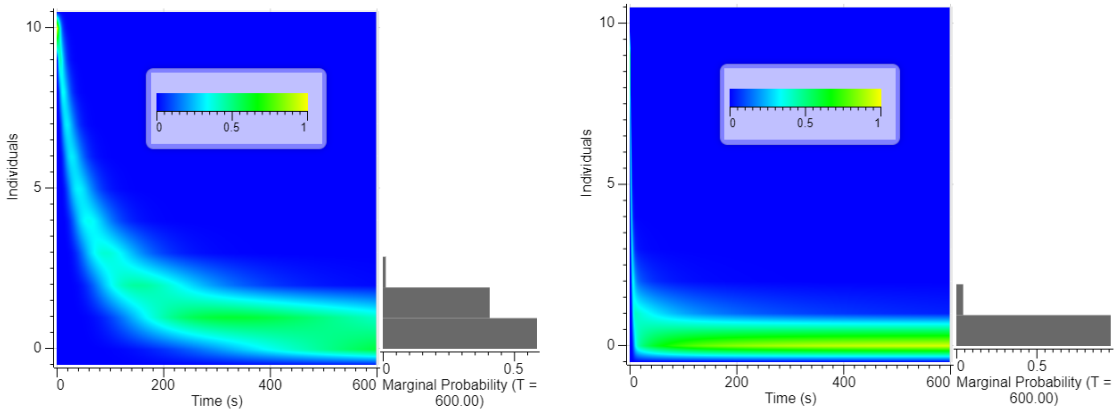
A comparison of stochastic, LNA and CME simulations is provided in Figure 14.




(a) CME simulation (b) Deterministic simulation

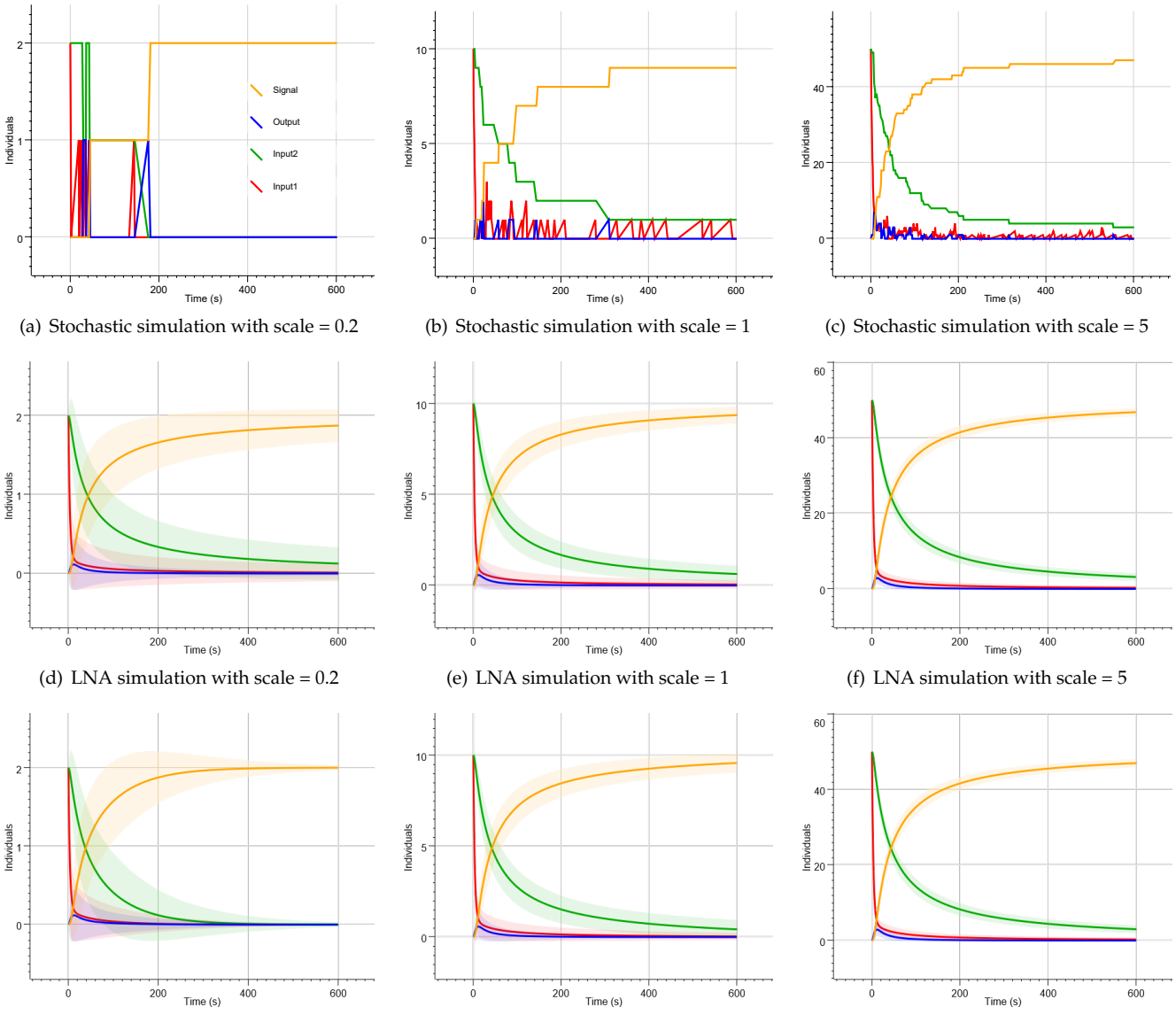



(c) Probability of Signal (d) Probability of Output




(e) Probability of Input2 (f) Probability of Input1

**Figure 13: Comparison of `cme` and `deterministic` simulation.**




(a) Stochastic simulation with scale = 0.2 (b) Stochastic simulation with scale = 1 (c) Stochastic simulation with scale = 5

(d) LNA simulation with scale = 0.2 (e) LNA simulation with scale = 1 (f) LNA simulation with scale = 5

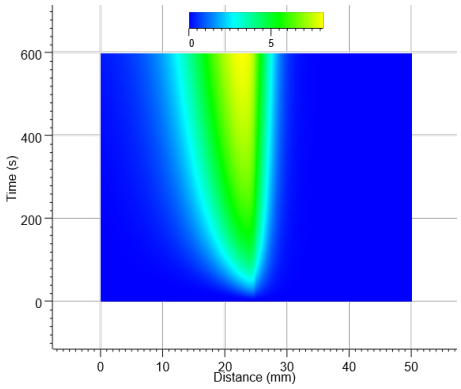
(g) CME simulation with scale = 0.2 (h) CME simulation with scale = 1 (i) CME simulation with scale = 5

**Figure 14: Effect of the number of molecules on simulation noise.** Simulation results are shown for stochastic, Linear Noise Approximation (LNA) and Chemical Master Equation (CME) simulations.

**A**

```
directive simulation {
  final=600;
  plots=[Signal];
}
directive simulator pde
directive spatial {
  boundary = ZeroFlux;
  dimensions = 2;
  diffusibles = [Input1 = 0.5; Input2 = 0.5];
  xmax = 50;
  nx = 101;
  dt = 1;
}
directive parameters [ k = 0.003; u = 0.1 ]
| 100 Reporter
| 100 Join
| Input1 = { spatial = { points =
    [ {x=0.3; y=0.3; width=0.4; value=10.0}]
  } }
| Input2 = { spatial = { points =
    [ {x=0.7; y=0.7; width=0.4; value=10.0}]
  } }
| Reporter + Output ->{k} sp2 + Signal
| Join + Input1 <->{k}{k} sp4 + sp3
| sp4 + Input2 <->{k}{k} sp1 + Output
```

**B**




**C**

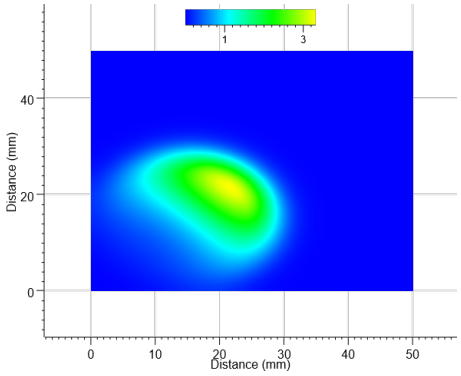



**Figure 15: Example with spatial simulation.** **A** Program code. We modify the running example (Figure 9) by replacing `10 Input1` and `10 Input2` with code to specify the initial placement of these two species. The `Input1` species is initialised with a value of 10 in a circular region of width 0.4 located at coordinates (0.3,0.3), while the `Input2` species is initialised in a similar fashion at coordinates (0.7,0.7). We also include additional spatial directives, which specify zero flux boundary conditions, and that the `Input1` and `Input2` species diffuse at rate 0.5. The spatial domain is over two dimensions, and is a square of edge 50mm, with a grid size of 100 and a simulation time step of 1s. **B** Simulation results of (A) using the 1 dimensional PDE solver by setting `dimensions = 1`. **C** Simulation results of (A) using the 2 dimensional PDE solver.

## 3.6 Spatial Simulation

We also provide simple capabilities for simulating spatio-temporal dynamics for CRNs, when these can be described by reaction-diffusion equations. The `pde` simulator has solvers for 1d and 2d spatial domains, and uses a Crank-Nicolson scheme to integrate the reaction-diffusion equations over time. The time series plot shows heatmaps for spatial simulator outputs, with $x$ position and time on the axes for 1d (Figure 15A), and $x$ position and $y$ position on the axes for 2d, with the plot updating at each output time (Figure 15B).

### 3.6.1 Spatial Simulation Method

By specifying `directive simulator pde`, a CRN system is converted to reaction-diffusion (RD) equations, which are an extension of the deterministic rate equations that includes isotropic diffusion:

$$\frac{\partial \mathbf{c}}{\partial t} = \mathbf{f}(\mathbf{c}) + D\nabla^2\mathbf{c} \tag{7}$$

Here, $\nabla^2$ is the laplacian operator, which describes the sum of the second partial derivatives over each spatial coordinate. In 1d, $\nabla^2 = \frac{\partial^2}{\partial x^2}$, whereas in 2d, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. The vector function $\mathbf{f}$ contains precisely the deterministic rate equations described above, e.g. $\mathbf{f}(\mathbf{c}) = S.\mathbf{v}(\mathbf{c})$.

In 2d, the RD equations are considered in a square domain (on a line in 1d), with $(x, y) \in [0, L] \times [0, L]$. Either periodic or Neumann (zero-flux) boundary conditions can be imposed (see below). For both 1d and 2d problems, the RD equations are solved numerically using the Crank-Nicolson method. A detailed derivation of the update schemes is presented in appendix B. In using this method, we are limited to solving on a square regular grid (spacing $\Delta x$), with a fixed time-step $\Delta t$.

### 3.6.2 Spatial settings

The solver settings can be changed using `directive spatial`.

| | |
|---|---|
| *Spatial*⟨*S*⟩ ::= { | Spatial record |
| `diffusibles` = *Map*⟨*S*, *Value*⟩; | Assigns diffusion rates to species ([]) |
| `default_diffusion` = *Float*; | Default diffusion rate for all species (0) |
| `dimensions` = *Integer*; | Number of spatial dimensions (1) |
| `random` = *Float*; | Relative random perturbation to initial conditions (0) |
| `xmax` = *Float*; | Maximum length of spatial domain (1) |
| `nx` = *Integer*; | Number of spatial grid points, including boundary (20) |
| `dt` = *Float*; | Time step for spatial simulations ($\frac{0.2dx^2}{maxdiffusibles}$) |
| `boundary` = *Boundary*; | Boundary conditions for spatial simulation (`Periodic`) |
| } | |

| | |
|---|---|
| *Bloundary* ::= | |
| \| *Periodic* | Periodic boundary conditions |
| \| *ZeroFlux* | Zero flux boundary conditions |

### 3.6.3 Spatial initial conditions

The initial conditions are specified using the attribute syntax for species initialisation. A `spatial` field can be assigned, which takes a record as argument. This record contains three fields that correspond to *generators* of initial conditions for the species in question. A `centralcore` creates a circle at the centre of the domain with a specified width, then separate concentrations can be assigned to the internal and external regions. A `point` is a circle at a specified $(x, y)$ position with a specified width and internal concentration. Multiple points can be assigned to the points field, using a list syntax. To apply the `random` setting, all centralcore and points are summed to any non-spatial initial condition assignment, and then the values at each grid-point are corrupted by a multiplicative perturbation with strength $r$, the value assigned to the `random` field. This perturbation is defined by drawing a uniform random number $u \sim \mathcal{U}(0, 1)$, and perturbing species $S$ at co-ordinates $x = x_i, y = y_j$ according to

$$s_{i,j} \max\{0, \rightarrow s_{i,j} \times (1 + r * (u - 0.5))\} \tag{8}$$

| | |
|---|---|
| *Placement* ::= { | Placement record |
| `centralcore` = *Core*; | Central core |
| `points` = `List`⟨`Point`⟩; | A list of points ([]) |
| `random` = *Float*; | Add random noise to initial population (0) |
| } | |

| | |
|---|---|
| *Core* ::= { | Core record |
| `width` = *Float*; | Diameter of central core, in relative units (0) |
| `inner` = *Float*; | Concentration inside central core (0) |
| `outer` = *Float*; | Concentration outside central core (0) |
| } | |

| | |
|---|---|
| *Point* ::= { | Point record |
| `x` = *Float*; | Relative $x$ position of point (0) |
| `y` = *Float*; | Relative $y$ position of point (0) |
| `width` = *Float*; | Diameter of point, in relative units (0) |
| `value` = *Float*; | Concentration of species inside point (0) |
| } | |

An example with spatial simulation is shown in Figure 15.

## 3.7 Inference

The inference module enables *parameter* values of a model to be inferred from observation data. Markov chain Monte Carlo (MCMC) is the methodology used, as implemented in the Filzbach software (https://github.com/predictionmachines/Filzbach). Filzbach uses a variation of the Metropolis-Hastings (MH) algorithm to perform Bayesian parameter inference. The MH algorithm is used to approximate the posterior probability of a parameter set from a hypothesised model taking on certain values, constrained by a likelihood function. The probability of each parameter value is then approximated by constructing a Markov chain of sampled parameter sets, such that a proposed parameter set is accepted with some probability, based on the ratio of the likelihood function evaluated at current and proposal parameter sets. A burn-in phase is used to enable the algorithm to determine suitable coefficients for the proposal distributions of each parameter. Burn-in samples are not stored in the list of posterior samples, but are instead discarded. For more information on MCMC methods, see [22]. MCMC methods, such as simulated annealing, have also been shown to efficiently find solutions to combinatorial optimisation problems [23], taking a stochastic search approach similar to the MH algorithm. Stochastic search can provide benefits over gradient-based optimisers by maintaining a nonzero probability of making up-hill moves, protecting against getting stuck in poor local optima. Eventually, an MCMC algorithm will *converge*, and the samples of the chain will represent the joint posterior distribution $P(\theta|\text{data})$. As such, the more samples that are used, the more accurately will the samples numerically approximate this distribution.

### 3.7.1 The likelihood function and noise model

The likelihood function is formed by taking all data specified in the program, and comparing corresponding simulations in a probabilistic way. Each data-point contributes a term to the likelihood function. Each are assumed to be independent and Gaussian distributed, centred on the corresponding simulated value and with some variance $\sigma^2$. As there are potentially multiple datasets, each with potentially multiple input treatments and output species, there could be several time-courses being bundled into this single function. Nevertheless, their composition is straightforward. For each time-series with measured values $\eta_k$ corresponding to times $t_k$, $k = 1, \ldots, n$, corresponding simulated values $y_k$ are obtained. Then, the likelihood function is defined as

$$L(\theta|\text{data}) = \prod_{\text{datasets, inputs, outputs, } k} \left\{ P(\eta_k; y_k, \sigma^2) \right\} \tag{9}$$

A *noise model* can be applied to the variance $\sigma^2$. We currently support two such models. The default model is a *constant* noise model, in which the variance is independent of signal intensity, and is treated as an inference parameter. An alternative *proportional-error* model can also be specified, which assumes that $\sigma^2 = \alpha \times y_k$, with $\alpha$ treated as an inference parameter. Since each species being measured may have different noise characteristics, the noise parameters can be specialised to each species, or assumed to be homogenous.

### 3.7.2 Inference settings

Inference has a number of settings that can be used to adapt the parameter inference problem. Most of these settings are propagated directly to the underlying Filzbach library. The inference settings can be specified as record entries of the `inference` directive. The definition of inference settings is given below, where default values are shown in parentheses.

| | |
|---|---|
| *Inference* ::= { | Inference record |
| `name` = *Name*; | Name of inference run (`default`) |
| `burnin` = *Integer*; | Number of discarded initial iterations (100) |
| `samples` = *Integer*; | Number of samples stored in posterior (100) |
| `thin` = *Integer*; | Ratio of posterior samples to discard (10) |
| `noise_model` = *Noise*; | Noise model (`constant`) |
| `prune` = *Boolean*; | Remove reactions involving species that are never produced (`false`) |
| `seed` = *Integer*; | Seed of random number generator (0) |
| `timer` = *Boolean*; | Collect timing statistics for simulations (`false`) |
| `partial` = *Boolean*; | Only run simulations affected by changed parameters (`false`) |
| } | |

| | |
|---|---|
| *Noise* ::= | |
| \| `constant` | Error is constant |
| \| `proportional` | Error is proportional to measured value |

### 3.7.3 Priors

A parameter can be supplied with an optional prior, which is used during inference to determine how parameter sampling is performed. If no prior is provided then the parameter is assumed to be fixed. The definition of priors is given below, where default values are shown in parentheses.

| | |
|---|---|
| *Prior* ::= { | Prior record |
| `interval` = *Interval*; | Scale used to vary the parameter |
| `distribution` = *Distribution*; | Prior distribution of the parameter |
| `variation` = *Variation*; | Method used to vary the parameter |
| } | |
| | |
| *Interval* ::= | |
| \| *Real* | Vary the parameter on a linear scale |
| \| *Log* | Vary the parameter on a logarithmic scale |
| | |
| *Distribution* ::= | |
| \| `Uniform`($Float_1$, $Float_2$) | Uniform distribution between $Float_1$ and $Float_2$ |
| \| `Normal`($Float_1$, $Float_2$) | Normal distribution, mean $Float_1$, standard deviation $Float_2$ |
| \| `TruncatedNormal` ($Float_1$, $Float_2$, $Float_3$, $Float_4$) | Truncated Normal distribution, mean $Float_1$, standard deviation $Float_2$, between $Float_3$ and $Float_4$ |
| | |
| *Variation* ::= | |
| \| *Random* | Randomly vary parameter |
| \| *Fixed* | Keep parameter fixed |
| \| *Initial* | Use default value as initial estimate |
| \| *Multiple* | Use a unique parameter in each sweep instance |

### 3.7.4 Datasets

Multiple time-courses with different time-points must be imported as separate files. When multiple species are measured simultaneously, their columns must be placed adjacent to another, in the order that is specified in the `plots` field of `directive simulation`. If multiple measurements are conducted simultaneously (for example, direct repeats, or responses to different treatments), blocks of measurements (potentially containing multiple species) can be placed adjacently to one another, so long as they can use the same time column, as mentioned earlier. To ensure that the simulations are compared with the correct columns, the sweeps must be defined so that the ordering of simulation instances is the same as the ordering of measurement columns in the data file.

# 4 Visual DSD Language

This section presents a detailed description of the Visual DSD language, following the brief language overview presented in Section 2.2. We illustrate the main features of the language by making a series of modifications to the running example from Section 2.2.1, reproduced for convenience in Figure 16A. Subsequent modifications to the running example are highlighted in grey.

## 4.1 DSD Programs

A *Program* in the Visual DSD language consists of multiple *Settings*, followed by multiple *Declarations*, followed by a *Process*, where the *Settings* are defined in Section 4.3.

| | |
|---|---|
| *Program* ::= *Settings Declarations Process* | DSD program |
| *Settings* ::= `directive` $Setting_1 \ldots$ `directive` $Setting_N$ | Multiple settings |
| *Declarations* ::= $Declaration_1 \ldots Declaration_N$ | Multiple declarations |

A *Declaration* can be for a module, a variable, or a domain, where a module declaration associates a module *Name* and multiple arguments $(Name_1, \ldots, Name_N)$ with a *Process*, a variable declaration associates a variable *Name* with a *Value*, and a domain declaration associates a domain *Identifier* with a set of domain *Properties*. An *Identifier* uniquely identifies a domain and can be a *Name* or a positive number.

| | |
|---|---|
| *Declaration* ::= | |
| \| `def` $Name(Name_1, \ldots, Name_N) = Process$ | Module declaration |
| \| `def` *Name* = *Value* | Variable declaration |
| \| `dom` *Identifier* = *Properties* | Domain declaration |

The domain *Properties* include the sequence `seq`, the `colour`, the `bind` rate, the `unbind` rate, and a list of consecutive `subdomains` that make up the domain. If a given domain property is not specified explicitly then a default property is used. In particular, if no sequence is provided then a unique sequence is automatically assigned to the domain from a library of sequences. If no colour is provided then a unique colour is automatically assigned to toehold domains, while all recognition domains are represented in grey. The default bind and unbind rates are 0.0003 and 0.1126, respectively, and the list of subdomains is assumed to be empty by default.

| | |
|---|---|
| *Properties* ::= { | Domain properties record (with default values) |
| `seq` =*Name*; | DNA sequence (assigned by compiler) |
| `colour` =*String*; | Colour (assigned by compiler) |
| `bind` =*Value*; | Binding rate (0.0003) |
| `unbind` =*Value*; | Unbinding rate (0.1126) |
| `subdomains` =*List*⟨*Domain*⟩; | Consecutive subdomains ([]) |
| } | |

An example with nucleotide sequences is shown in Figure 16B and an example with subdomains is shown in Figure 17.

A *Process* can be an *Initial* species, a user-defined *Reaction*, a domain *Identifier* unique to a *Process*, a module *Name* instantiated with multiple values, or multiple processes in parallel, separated by the parallel composition operator (`|`).

| | |
|---|---|
| *Process* ::= | |
| \| *Initial*⟨*Species*⟩ | Initial species |
| \| *Reaction*⟨*Species*⟩ | User-defined reaction |
| \| `new` *Identifier* = *Properties Process* | Domain with properties unique to a process |
| \| `new` *Identifier Process* | Domain with default properties unique to a process |
| \| $Name(Value_1, \ldots, Value_N)$ | Module instance |
| \| $(Process_1 \mid \ldots \mid Process_N)$ | Multiple parallel processes |

The syntax of an *Initial* species and a user-defined *Reaction* is the same as the corresponding syntax in the Visual CRN language, defined in Section 3.2, except that a DSD *Species*, defined in Section 4.2, is used instead of a CRN *Species*.

Each user-defined *Reaction* is added to the set of automatically generated reactions and can encode behaviours that are not currently supported by the reaction enumeration process. For example, user-defined reactions can encode the effects of DNA or RNA enzymes, or hypotheses about the rates of interaction of particular species

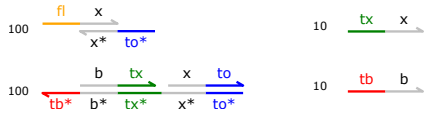
**A**

```
1  directive simulation {
2    final=600;
3    plots=[Input1();Input2();Output();Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  dom tb = {bind=k;unbind=u;colour="red"}
9  dom tx = {bind=k;unbind=u;colour="green"}
10 dom to = {bind=k;unbind=u;colour="blue"}
11 def Input1() = <tb^ b>
12 def Input2() = <tx^ x>
13 def Output() = <x to^>
14 def Join() = {tb^*}[b tx^]:[x to^]
15 def Reporter() = <fl^>[x]{to^*}
16 def Signal() = <fl^ x>
17 ( 10 Input1()
18 | 10 Input2()
19 | 0 Output()
20 | 100 Join()
21 | 100 Reporter()
22 | 0 Signal()
23 )
```

**D**

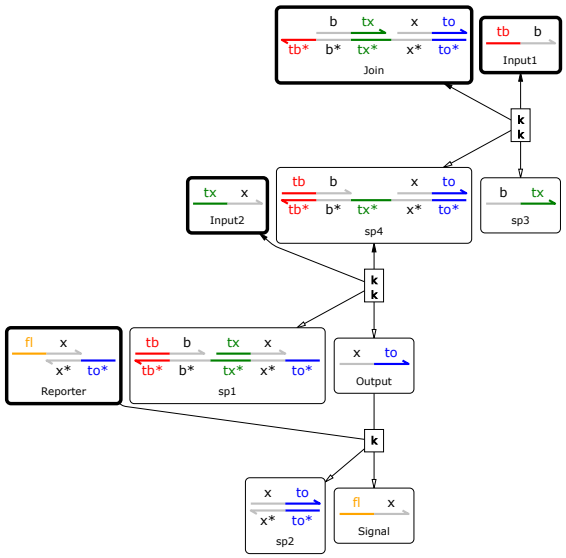
```
1  directive simulation {
2    final=600;
3    plots=[Input1();Input2();Output();Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  directive rendering {classic = {mode = nucleotides}}
9  dom tb = {bind=k;unbind=u;colour="red";seq=CTGCTA}
10 dom tx = {bind=k;unbind=u;colour="green";seq=TTCCAC}
11 dom to = {bind=k;unbind=u;colour="blue";seq=TCCTAC}
12 dom b = {seq=CATTGCTTCTACGAGTCATCC}
13 dom x = {seq=CATTGCACCTTAGAGTCCGAA}
14 dom fl = {seq=T}
15 def Input1() = <tb^ b>
16 def Input2() = <tx^ x>
17 def Output() = <x to^>
18 def Join() = {tb^*}[b tx^]:[x to^]
19 def Reporter() = <fl^>[x]{to^*}
20 def Signal() = <fl^ x>
21 ( 10 Input1()
22 | 10 Input2()
23 | 0 Output()
24 | 100 Join()
25 | 100 Reporter()
26 | 0 Signal()
27 )
```

**B**




**E**


**C**




**F**

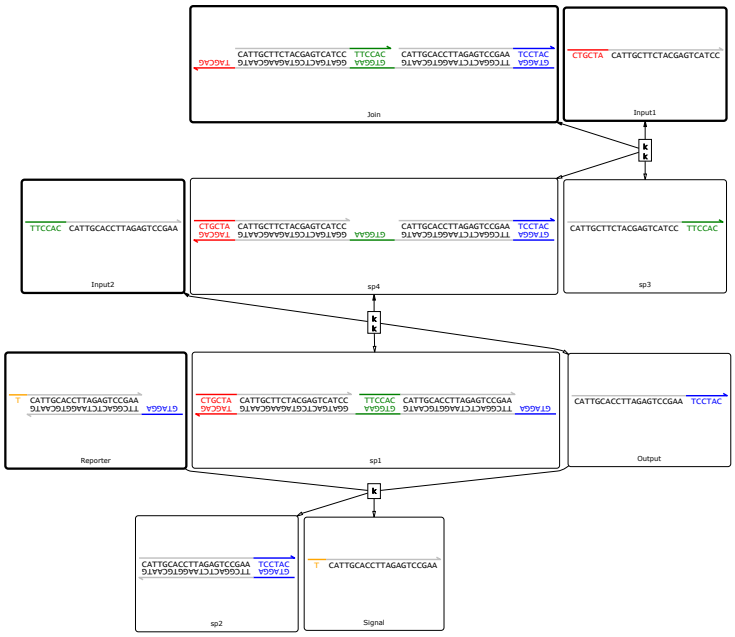
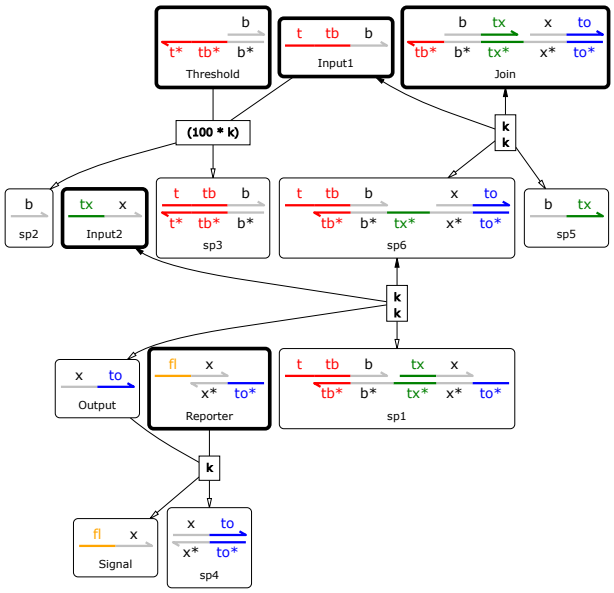



**Figure 16: Example with nucleotide sequences.** **A** Program code of the running example from Section 2.2.1. **B** Initial conditions generated from (A). **C** CRN generated from (A), where species that are present initially are outlined in bold. **D** Program code of an example with nucleotide sequences, where differences from the running example are highlighted in grey. Short sequences are assigned to toehold domains `tb`, `tx` and `to` (lines 9-11), while longer sequences are assigned to recognition domains `b` and `x` (lines 12-13). The toehold domain `fl` does not correspond to a physical DNA sequence, but instead represents a fluorophore. However, since the Visual DSD language does not currently support the representation of fluorophores, we use a domain assigned with the arbitrary DNA sequence `T` (line 14). To display the nucleotide sequences we write `directive rendering {classic = {mode = nucleotides}}` (line 8). **E** Initial conditions generated from (D). **F** Reaction graph generated from (D). The initials conditions and the reaction graph are the same as in (B-C) but with the domain names replaced by their corresponding DNA sequences.

**A**

```
1  directive simulation {
2    final=600;
3    plots=[Input1();Input2();Threshold();Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  dom tb = {bind=k;unbind=u;colour="red"}
9  dom tx = {bind=k;unbind=u;colour="green"}
10 dom to = {bind=k;unbind=u;colour="blue"}
11 dom t = {bind=k; unbind=u; colour="red"}
12 dom th = {bind=100*k; unbind=u; subdomains=[t;tb]}
13 def Input1() = <t^ tb^ b>
14 def Input2() = <tx^ x>
15 def Output() = <x to^>
16 def Join() = {tb^*}[b tx^]:[x to^]
17 def Threshold() = {t^* tb^*}[b]
18 def Reporter() = <fl^>[x]{to^*}
19 def Signal() = <fl^ x>
20 ( 10 Input1()
21 | 10 Input2()
22 | 0 Output()
23 | 100 Join()
24 | 2 Threshold()
25 | 100 Reporter()
26 | 0 Signal()
27 )
```

**B**




**C**

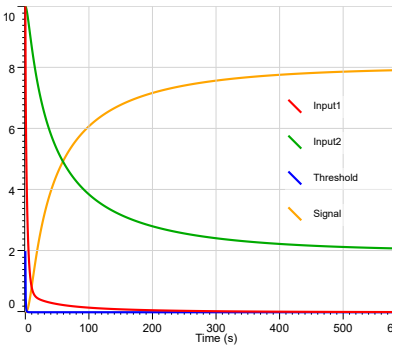



**Figure 17: Example with subdomains. (A)** Program code, where differences from the running example are highlighted in grey. Two new toehold domains `t` and `th` are defined (lines 11-12), where `th` has toeholds `t` and `tb` consecutively as subdomains, and the binding rate of these consecutive toeholds is assumed to be 100 times the binding rate of just the `tb` toehold on its own. The `Input1` strand is extended with the toehold `t` (line 13) and a new `Threshold` complex is defined (line 17), which contains a single-stranded region that is complementary to `t tb`. Essentially, the `Threshold` complex out-competes the `Join` complex for binding to the `Input1` strand, due to its longer region of complementarity to `Input1`. This approach can be used to consume input strands that are inadvertently produced, for example through leak reactions, in order to prevent any leaked inputs from generating a downstream output, up to a certain threshold [18]. In this example, we set the concentration of `Threshold` at 2nM (line 24) and plot this concentration over time instead of the concentration of `Output` (line 3). Note that currently the domain `th` cannot be used directly in the program code, since it is not expanded automatically into its consecutive subdomains. The implementation of this expansion is left for future work. **B** CRN generated from (A). We observe that the `Input1` strand can interact with the `Join` complex on toehold `tb` at rate `k`, and with the `Threshold` complex on consecutive toeholds `t tb` at rate `100*k`. **C** Simulation results for (A). We observe that the `Threshold` is consumed preferentially at the beginning of the simulation, and that the level of `Signal` is approximately 2nM less than the initial concentration of `Input1`, due to consumption of `Input1` by the `Threshold`.

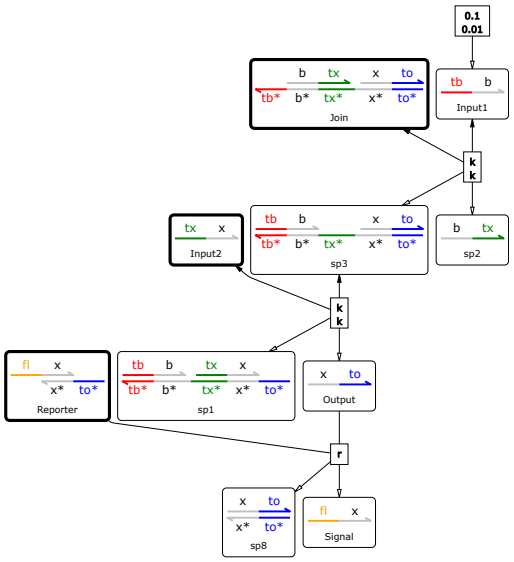
[7]. The products of a user-defined reaction can themselves potentially interact with other species to produce new reactions, and are therefore included in the reaction enumeration process. In cases where a user-defined reaction has the same reactants and products as an automatically generated reaction, the user-defined reaction *replaces* the automatically generated one. This allows custom kinetic rates to be defined for specific reactions. By default, the kinetic rate of a strand displacement reaction is determined by the toehold that mediates the displacement. However, in some cases we may wish to allow different interactions mediated by the same toehold to take place at different rates [8]. In these cases it is necessary to override the automatically generated rate with a reaction-specific one. If needed, the rate of every reaction can be manually overridden. In such cases, the automatic generation of reactions still provides a useful means of checking that no reactions have been inadvertently omitted. An example with user-defined reactions is shown in Figure 18.

The declaration of domains using `new` is similar to the declaration using `dom`, except that the domain *Identifier* is assumed to be unique to a given *Process*. As a result, if the same domain *Identifier* is declared multiple times

**A**

```
1  directive simulation {
2    final=600;
3    plots=[Input1();Input2();Output();Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1;r=0.01]
7  directive compilation infinite
8  dom tb = {bind=k;unbind=u;colour="red"}
9  dom tx = {bind=k;unbind=u;colour="green"}
10 dom to = {bind=k;unbind=u;colour="blue"}
11 def Input1() = <tb^ b>
12 def Input2() = <tx^ x>
13 def Output() = <x to^>
14 def Join() = {tb^*}[b tx^]:[x to^]
15 def Reporter() = <fl^>[x]{to^*}
16 def Signal() = <fl^ x>
17 ( 0 Input1()
18 | ->{0.1} Input1()
19 | Input1() ->{0.01}
20 | Output() + Reporter() ->{r} [x to^] + Signal()
21 | 10 Input2()
22 | 0 Output()
23 | 100 Join()
24 | 100 Reporter()
25 | 0 Signal()
26 )
```

**B**




**C**

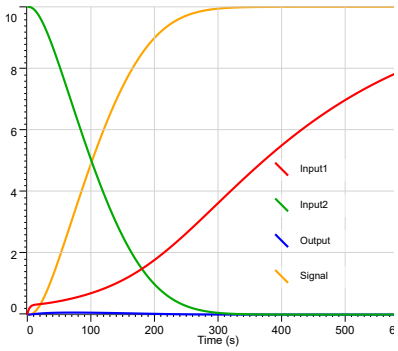



**Figure 18: Example with user-defined reactions. A** Program code, where differences from the running example are highlighted in grey. An additional kinetic parameter `r` is declared (line 6) and used to constrain the rate of interaction between the `Output` and the `Reporter` (line 20). By default, the rate of this interaction is equal to the rate `k` associated with the toehold `to` that mediates the interaction. However, in practice we may wish to allow different interactions mediated by the same toehold to take place at different rates. We also use reactions to model the assumption that the `Input1` strand is produced at rate 0.1 and consumed at rate 0.01 by enzymatic synthesis and degradation (lines 18 - 19). Since Visual DSD does not support an explicit representation of enzymes, their effects can be modelled at an abstract level using reactions. **B** CRN generated from (A). We observe that the rate of interaction between the `Output` and the `Reporter` has been set to `r`, and that the production and degradation reactions for `Input1` have been added to the generated CRN. **C** Simulation results for (A). We observe that the concentration of `Input1` increases over time, since new strands are continually being produced by the additionally specified reactions.

then it will be automatically renamed each time to ensure that it is globally unique. In principle, a process using `new` can always be converted to an equivalent process using `dom`. However, this places a burden on the user to ensure that a globally unique *Identifier* is used for each `dom` declaration. The use of `new` is particularly helpful when implementing translation schemes, such as those that convert a high-level chemical reaction network to a Visual DSD program. For example, in [6] a collection of modules was defined such that each module represented a strand displacement encoding of a high-level chemical reaction. The encoding required intermediate DNA species to be created using domains that were unique to each reaction. The use of `new` allowed these unique domains to be created automatically instead of being individually named by the user. This allowed modules to be directly composed without the need for global renaming. An example with domains unique to a process is shown in Figure 19.

**A**

```
1  directive simulation {
2    final=600;
3    plots=[<tb^ b>;<tx^ x>;<x to^>;<fl^ x>];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  dom tb = {bind=k;unbind=u;colour="red"}
9  dom tx = {bind=k;unbind=u;colour="green"}
10 dom to = {bind=k;unbind=u;colour="blue"}
11 def Input1(b) = <tb^ b>
12 def Input2(x) = <tx^ x>
13 def Output(x) = <x to^>
14 def Join(b,x) = {tb^*}[b tx^]:[x to^]
15 def Reporter(x) = <fl^>[x]{to^*}
16 def Signal(x) = <fl^ x>
17 def System() =
18   new b new x
19   ( 10 Input1(b)
20   | 10 Input2(x)
21   | 0 Output(x)
22   | 100 Join(b,x)
23   | 100 Reporter(x)
24   | 0 Signal(x)
25   )
26 ( System() | System() )
```

**B**

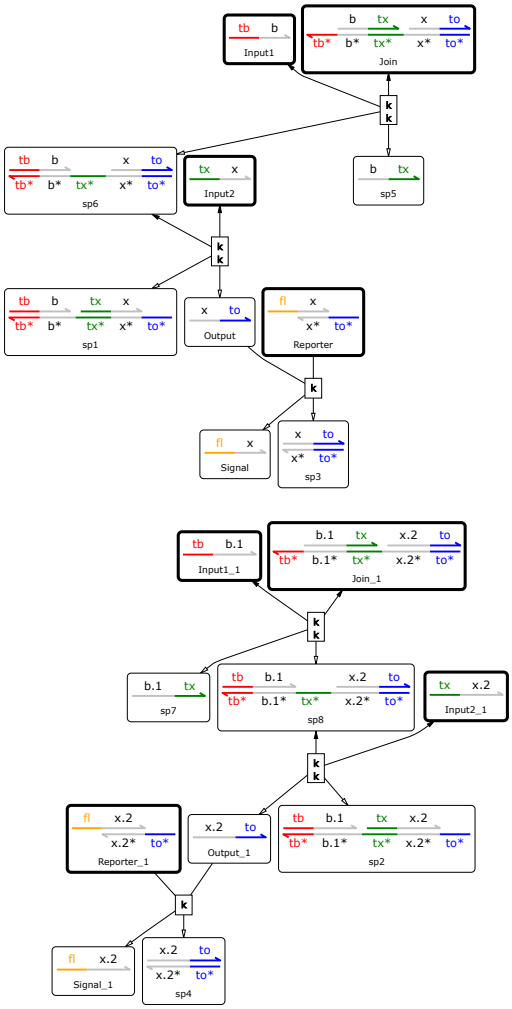



**Figure 19: Example with domains unique to a process.** **A** Program code, where differences from the running example are highlighted in grey. Each module definition is updated by adding the recognition domains used within the module as arguments (lines 11-16). An additional `System` module is defined, which declares two recognition domains `b` and `x` that are unique to the module, and then instantiates the remaining modules with these domains as arguments (lines 17-24). In addition, the `plots` are updated by replacing any module instances with their corresponding DSD species (line 3). This allows the plots to match all occurrences of domains defined using `new`. Finally, two copies of the `System` module are instantiated in parallel (line 26). Note that an equivalent program could be defined by declaring distinct domains `b`, `x`, `b1`, `x1` globally, and then using `b`, `x` in the first system and `b1`, `x1` in the second. **B** CRN generated from (A). The Visual DSD compiler has automatically generated unique names `b.1`, `x.1` for the second system, resulting in two distinct CRNs, one for each system. Note that the toehold domains `tb`, `tx`, `to` are still shared between the two systems.

## 4.2 DSD Species

A *Species* in the Visual DSD language can be a *Strand*, a *Complex*, a *Tile*, or a module *Name* instantiated with multiple values:

| *Species* ::= | |
|---|---|
| \| *Strand* | Single DNA strand |
| \| *Complex* | Complex consisting of multiple DNA strands bound to each other |
| \| *Tile* | Tile containing tethered species |
| \| $Name(Value_1, \ldots, Value_N)$ | Module instance |

The remainder of this section presents definitions for the different types of species, together with illustrative examples.

### 4.2.1 Strands

A *Strand* represents a single-stranded DNA molecule, consisting of a *Sequence* of domains enclosed in either angle brackets or curly braces:

| *Strand* ::= | |
|---|---|
| \| <*Sequence*> | Upper strand |
| \| {*Sequence*} | Lower strand |

A *Sequence* enclosed in angle brackets represents a strand with the 3′ end on the right, also referred to as an Upper strand, while a *Sequence* enclosed in curly braces represents a strand with the 3′ end on the left, also referred to as a Lower strand. Strands are identical up to rotation symmetry, such that an Upper strand with a given sequence is identical to a Lower strand with this sequence reversed. This captures the fact that the same physical DNA strand can be represented textually in two different orientations: from left to right and from right to left.

A *Sequence* can be a *Domain*, a domain complement represented by appending the (*) character to the domain, a `tether` with multiple location tags, or multiple sequences separated by spaces:

| *Sequence* ::= | |
|---|---|
| \| *Domain* | Domain |
| \| *Domain*$^*$ | Domain complement |
| \| `tether`$(Name_1, \ldots, Name_N)$ | Tether with multiple location tags, $N \geq 1$ |
| \| $Sequence_1 \ldots\ Sequence_N$ | Multiple sequences, $N \geq 1$ |

A strand is assumed to be tethered if it contains at least one `tether` in its sequence, and a species is assumed to be tethered if it contains at least one tethered strand.

A *Domain* can be a recognition domain represented by an *Identifier*, a toehold domain represented by appending the (^) character to the identifier, or a Wildcard domain, which matches any domain and is only used for plotting:

| *Domain* ::= | |
|---|---|
| \| *Identifier* | Recognition domain |
| \| *Identifier*$^\wedge$ | Toehold domain |
| \| _ | Wildcard, used for plotting only |

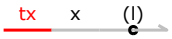
Recognition domains are assumed to be long enough to bind irreversibly to their complement, while toehold domains are assumed to be short enough to bind reversibly to their complement. We assume that only species inside the list of `plots` in the `simulation` settings can contain a Wildcard domain _. This matches a single domain, causing multiple species to be plotted. For example, the pattern `<_ x>` matches `<tx^ x>` and `<tb^ x>` but not `<t^ tx^ x>`.

For examples of single strands, consider the program in Figure 16A. The strand `<tx^ x>` consists of the toehold domain `tx^` following by the domain `x`, while the strand `<tb^ b>` consists of the toehold domain `tb^` following by the domain `b`. These strands are represented graphically as follows:


They are called Upper strands since their 3′ end is on the right and they are represented graphically on the upper region of a double-stranded complex. Since strands are identical up to rotation symmetry, they are also identical to the strands `{x tx^}` and `{b tb^}`, respectively. These strands are represented graphically as follows:

They are called Lower strands since their 3′ end is on the left and they are represented graphically on the lower region of a double-stranded complex.

We can also add tethers to a strand, associated with one or more location tags. The strand `<tx^ x tether(l)>` adds a tether to the 3′ end of strand `<tx^ x>`, associated with location tag `l`. The tethered strand is represented graphically as follows:




The tethers denote attachment points for a strand to a tile, and the location tags associated with a tether encode hypotheses about whether tethered strands are physically close enough to interact with each other, without the need to explicitly define the geometry of the strands [10]. Additional explanations for species tethered to a tile are described in Section 4.2.3.

### 4.2.2 Complexes

A *Complex* can be one or more *Segments*, connected using the connection operator *O*. It can also be a *Left* hairpin connected to multiple segments on the right, a *Right* hairpin connected to multiple segments on the left, or a *Left* hairpin connected to multiple segments followed by a *Right* hairpin:

*Complex* ::=

| | |
|---|---|
| $\mid$ *Segment* $O_1$ *Segment*$_1 \ldots O_N$ *Segment*$_N$ | Segments |
| $\mid$ *Left* $O_1$ *Segment*$_1 \ldots O_N$ *Segment*$_N$ | Segments with left hairpin |
| $\mid$ *Segment*$_1$ $O_1 \ldots$ *Segment*$_N$ $O_N$ *Right* | Segments with right hairpin |
| $\mid$ *Left O Segment*$_1$ $O_1 \ldots$ *Segment*$_N$ $O_N$ *Right* | Segments with left and right hairpins |

The connection operator *O* can connect a segment along its lower strand (:) or along its upper strand (::):

*O* ::=

| | |
|---|---|
| $\mid$ : | Connection along lower strand |
| $\mid$ :: | Connection along upper strand |

A *Segment* is a double stranded region of DNA, referred to as a duplex, with single-stranded *Overhangs* to the left and right. A *Left* hairpin is a duplex with a hairpin loop to the left and single-stranded overhangs to the right, while a *Right* hairpin is a duplex with a hairpin loop to the right and single-stranded overhangs to the left:

| | |
|---|---|
| *Segment* ::= *Overhangs* [*Sequence*] *Overhangs* | Duplex with left and right overhangs |
| *Left* ::= <*Sequence*}[*Sequence*]*Overhangs* | Left hairpin with right overhangs |
| *Right* ::= *Overhangs*[*Sequence*]{*Sequence*> | Right hairpin with left overhangs |

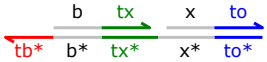
*Overhangs* can be a single strand, an upper and a lower strand, or empty:

*Overhangs* ::=

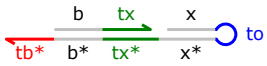
| | |
|---|---|
| $\mid$ *Strand* | Single strand overhang |
| $\mid$ {*Sequence*$_1$} <*Sequence*$_2$> | Lower and upper strand overhangs |
| $\mid$ <*Sequence*$_1$> {*Sequence*$_2$} | Upper and lower strand overhangs |
| $\mid \emptyset$ | No overhangs |

If both lower and upper strand overhangs are present, the order of the strands is not significant.

For examples of complexes, consider the program from Figure 16A. The complex `{tb^*}[b tx^]:[x to^]` is formed by connecting two segments. The first segment `{tb^*}[b tx^]` represents a duplex consisting of the strand `<b tx^>` bound to its complementary strand `{b* tx^*}`, with a lower strand overhang `{tb^*}` to the left of the duplex. The second segment `[x to^]` represents a duplex consisting of the strand `<x to^>` bound to its complementary strand `{x* to^*}`. These two segments are connected along the bottom strand by the connection operator (`:`). Although the textual representation of complexes is defined as a connection of segments, in reality the connection results in a single continuous bottom strand, as shown in the graphical representation of the complex:




The above complex can be modified to contain a right hairpin, resulting in the complex `{tb^*}[b tx^]:[x]{to^>`. The complex contains a right hairpin `[x]{to^>` with stem `[x]` and loop `{to^>` and is represented graphically as follows:




### 4.2.3 Tiles

A *Tile* contains multiple tethered *Species* that cannot themselves be tiles, and is represented by enclosing the set of tethered species in double brackets:

*Tile* ::= $[[Species_1|\ldots|Species_N]]$ Each *Species* is tethered and cannot be a Tile, $N \geq 1$

The tile represents a surface such as a DNA origami, to which species are attached using tethers. Each species in a tile contains at least one tether, and each tether is associated with one or more location tags. We assume that all of the species that share a particular tag are tethered sufficiently close to each other to interact, and that two tethered species can only interact if they have at least one tag in common. We also associate each tag with a *local concentration*, which denotes the fact that tethered species may interact at a faster rate compared to species in solution. For example, to associate locations tags `l1` and `l2` with local concentrations `c1` and `c2` we write `directive locations [l1 = c1; l2 = c2]`. An example with species tethered to a tile is shown in Figure 20.

## 4.3 DSD Settings

The settings for the Visual DSD language are summarised below, together with their default values. The settings also include the Visual CRN settings, instantiated with the Visual DSD *Species*:

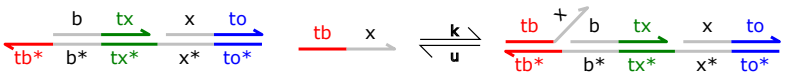
*Setting* ::=

| | |
|---|---|
| `compilation` *Compilation* | Compilation method (`default`) |
| `unproductive` | Enable unproductive reactions |
| `jit` | Enable just-in-time simulation |
| `leaks` | Enable leak reactions |
| `declare` | Require all domains to be declared |
| `polymers` | Allow polymers to form |
| `locations` *Map*⟨*Name*, *Value*⟩ | Assign local concentrations to location tags ([]) |
| `rendering` *Rendering* | Strand visualization mode |
| `leak` *Value* | Global rate of leak reactions (1e-9) |
| `tau` *Value* | Global rate of fast reactions (0.1126) |
| `migrate` *Value* | Global rate of migration per nucleotide (8000) |
| `lengths` *Integer Integer* | Default toehold and recognition domain lengths, respectively (6, 20) |
| `toeholds` *Float Float* | Default toehold binding and unbinding rates, respectively (0.0003, 0.1126) |
| *Crn.Setting*⟨*Species*⟩ | CRN setting |

We briefly describe each setting below.

**compilation** sets the method that is used to compile a Visual DSD program to a CRN (see Section 4.3.1).

**unproductive** enables the generation of *unproductive* reactions, which involve spurious toehold binding and unbinding reactions that are not able to trigger a subsequent displacement reaction. Since these reactions increase the computational cost of compilation and simulation, they are disabled by default. An example of an unproductive reaction is the following:




The reaction is unproductive since the toehold binding does not give rise to any subsequent reaction other than an immediate unbinding. This is because the domain `x` does not match the domain `b` and therefore cannot cause a displacement. In practice, removing these unproductive reactions often has little effect on model simulation, however in some cases these effects are non-negligible, depending on the concentration of species in solution. Note that unproductive is not possible for the `infinite` compilation method (see Section 4.3.1).

**jit** enables just-in-time simulation, by generating the CRN dynamically during simulation instead of of up-front before the simulation is started. This is necessary for systems for which the CRN has a potentially unbounded number of reactions. Note that jit is currently only supported for stochastic simulation.

**leaks** enables the generation of *leak* reactions, which are spurious strand displacement reactions initiated without a toehold, and take place at much lower rates than toehold-mediated strand displacement reactions (see Section 4.3.3).

**A**

```
1  directive simulation {
2    final=600;
3    plots = [Input1(); Input2(); Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  directive polymers
9  directive locations [l=10000]
10 dom tb = {bind=k;unbind=u;colour="red"}
11 dom tx = {bind=k;unbind=u;colour="green"}
12 dom to = {bind=k;unbind=u;colour="blue"}
13 def Input1() = <tb^ b>
14 def Input2() = <tx^ x>
15 def Output() = <x to^>
16 def Join() = {tether(l) tb^*}[b tx^]:[x]{to^>
17 def Reporter() = <fl^>[x]{to^* tether(l)}
18 def Signal() = <fl^ x>
19 ( 10 Input1()
20 | 10 Input2()
21 | 100 [[ Join() | Reporter() ]]
22 | 0 Signal()
23 )
```

**B**

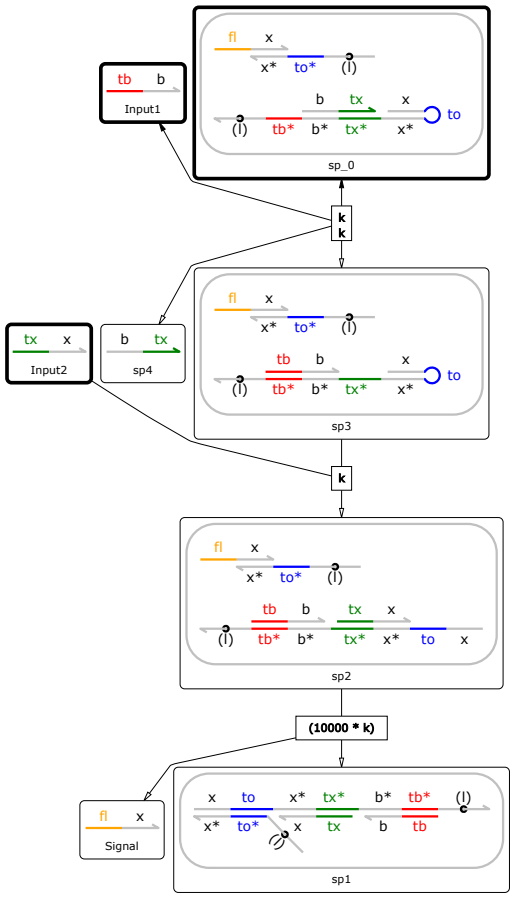



**Figure 20: Example with species tethered to a tile. A** Program code, where differences from the running example are highlighted in grey. The `Join` species is modified so that it contains a hairpin with domain `to^` in the loop and a tether with location tag `l` on the 3′ end of the `tb^*` overhang (line 16). In addition, the reporter is modified so that it contains a tether with the same location tag `l` on the 5′ end of the `to^*` overhang (line 17). The initial conditions are modified so that they contain 100nM of a tile with tethered `Join` and `Reporter` species (line 21). A local concentration of 10000nM is associated to the location tag `l` (line 9). This models the assumption that the two species are tethered close to each other, such that their effective concentration is substantially increased. In practice, these local concentrations can be estimated from data using parameter inference [13]. Since the two species can interact to form a polymer, `directive polymers` is needed to enable polymer formation (line 8). Polymer formation is disabled by default, since polymers can potentially be of unbounded length, which may result in the Visual DSD compiler getting stuck in an infinite loop. Enabling the formation of polymers means that such infinite loops could potentially occur. Finally, since no `Output` strand is produced, the plots are modified so that only `Input1`, `Input2` and `Signal` are plotted (line 3). **B** CRN generated from (A). The freely diffusing `Input1` strand binds to the `Join` complex tethered to the tile and displaces the `<b tx^>` strand. The freely diffusing `Input2` strand then opens the hairpin, exposing the `to^` toehold. This then binds to the exposed `to^*` toehold of the tethered `Reporter` complex and displaces the `Signal`. The interaction is scaled by the local concentration `l`, since the `Reporter` and `Join` complex are tethered in close proximity to each other. Importantly, the resulting scaled rate `10000*k` of this reaction is unimolecular with units $s^{-1}$, since it involves two complexes tethered to the same origami at fixed locations, and therefore the interaction between these two tethered complexes is not affected by the concentration of strands in solution.

**declare** allows the user to enable stricter syntax checking, by producing an error if a domain is used without having first been declared. This can be helpful when debugging large programs, since a mis-spelled domain can be hard to detect and can significantly alter the behaviour of the system.

**polymers** enables complexes to bind together to form polymers of potentially unbounded length.

**leak** sets the rate of all leak reactions (default is $10^{-9}$ $nM^1s^1$).

**tau** sets the rate of all fast reactions in the `finite` compilation method (default is 0.1126 $s^1$).

**migrate** sets the migration rate of a single nucleotide (default is 8000 $s^1$). The branch migration rate for a domain of length $L$ is given by $r/L^2$, where $r$ is the single nucleotide migration rate.

**lengths** sets default values for the lengths of toeholds and recognition domains. For the purpose of computing rate constants, all long domains are assumed to have the same length. For example, the code `directive lengths 5 15`

assigns a length of 5nt to toeholds and 15nt to recognition domains. The default values are 6nt for toeholds and 20nt for recognition domains. The value provided for toeholds must be greater than that provided for recognition domains or the system will raise an error. Currently the assigned default length for toehold domains is not used to calculate toehold binding and unbinding rates. Instead the user may set toehold binding and unbinding rates directly on a per-toehold basis. Note that specific nucleotide sequences are not currently used when computing rate constants. The implementation of this functionality is left for future work.

**toeholds** sets the default binding and unbinding rates, respectively, for domains which are declared without explicit rates. The default values are 0.0003nM$^1$s$^1$, for the binding rate and 0.1126 s$^1$ for the unbinding rate.

Additional explanations and examples for the settings are provided in the remainder of this section.

### 4.3.1 Compilation

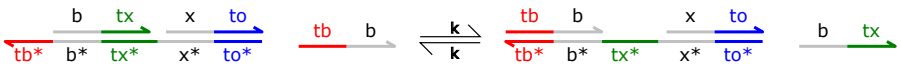
*Compilation* can be set to one of four methods, `infinite`, `default`, `finite` and `detailed`:

*Compilation* ::=

| | |
|---|---|
| `infinite` | Migration and unbinding are infinite |
| `default` | Migration is infinite |
| `finite` | Migration and unbinding are merged and finite |
| `detailed` | Migration and unbinding are finite |

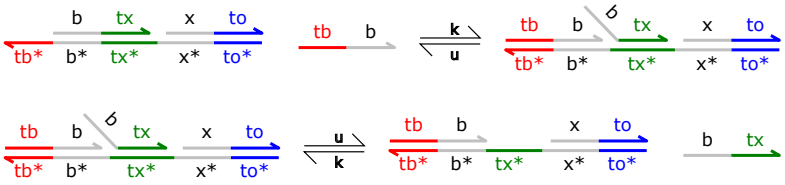
Each method specifies a different set of assumptions that are used when compiling a Visual DSD program to a CRN. If no compilation method is provided then the `default` method is used. The different methods are summarised below.

**Infinite** The concentration of species is assumed to be sufficiently low that the rates of unbinding and migration reactions are infinite compared to the rates of binding reactions. As a result, strand displacement is assumed to take place in single step that merges binding, migration and unbinding. Since branch migration is infinite, complexes are considered equal up to branch migration. An example of a one step strand displacement compiled using the infinite method is as follows:

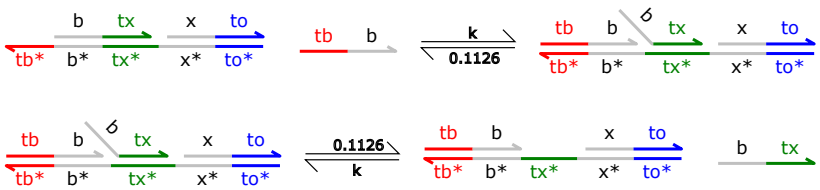



This method is used for the running example in Figure 16A. Note that unproductive reactions are not produced by this method, since the toehold will unbind immediately if no displacement is possible. This means that certain programming idioms such as cooperative strand displacement are not possible using this method.

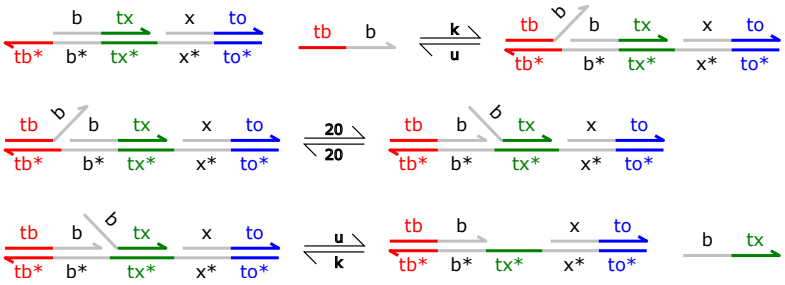
**Default** The concentration of species is assumed to be sufficiently low that the rate of migration reactions is infinite, however the rate of unbinding reactions is finite. As a result, strand displacement is assumed to take place in two steps, binding followed by unbinding. Since branch migration is infinite, complexes are considered equal up to branch migration. The unbinding rate is determined by the rate associated with the toehold on which the unbinding takes place. An example of a two step strand displacement compiled using the default method is as follows:




**Finite** The concentration of species is assumed to be sufficiently low that the rates of unbinding and migration reactions are fast but finite, where consecutive fast reactions are merged into a single reaction with rate `tau`. As a result, strand displacement is assumed to take place in two steps, binding followed by a fast step that merges migration and unbinding. Since branch migration is fast, complexes are considered equal up to branch migration. An example of a two step strand displacement compiled using the finite method is as follows:



**Detailed** This method is the most detailed and assumes that binding, migration and unbinding have finite rates. As a result, complexes are not considered equal up to branch migration. The branch migration rates are calculated from the migration rate per nucleotide and the number of nucleotides in the domain. An example of a three step strand displacement compiled using the detailed method is as follows:




Moving from the Infinite through to the Detailed compilation method causes the number of generated reactions to increase substantially. As a result, the computational cost of compilation also increases. An example with the same system compiled using the different methods is shown in Figure 21.

The CRN generated from a given Visual DSD program can vary substantially depending on the compilation method used. However, in cases where the concentration of species in solution is sufficiently low, the binding rate is the limiting step, such that the additional reactions for migration and unbinding to not substantially affect the system kinetics. An example with the same system compiled using different methods and simulated deterministically is shown in Figure 22.

### 4.3.2 Just In Time Simulation

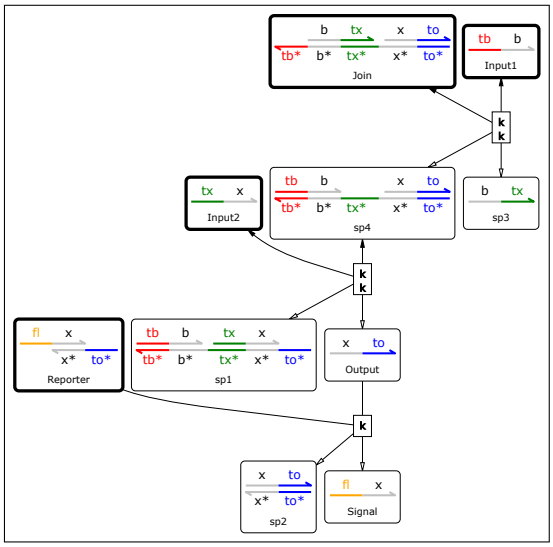
Enabling just-in-time simulation (JIT) allows the CRN of a Visual DSD program to be generated dynamically during the course of a simulation, rather than before the simulation is started. The rationale behind the JIT simulator is that some systems can become very large, with many thousands of possible reactions. This means that the full compilation process can take a very long time. This is particularly true for the larger example programs when leaks are enabled. However, since leak reactions have a low probability of actually happening we can spend a large amount of time computing reactions that will probably never happen during a particular simulation run.

The JIT simulator alleviates this problem by compiling new reactions dynamically during the simulation run. When the CRN button is clicked in JIT mode, only the Initial tab is populated with the initial species. The other output tabs are populated when the simulation is paused or reaches the end of its run. When the JIT simulator is running, it checks after each reaction to see if the products of that reaction have been seen before. If not, a single compilation step occurs which augments the system with the new species and the new reactions which are made possible by the introduction of those species. Thus the reaction graph and list of species is gradually built up over the course of a simulation run. Running the same program multiple times in JIT mode may produce a different final reaction graph, depending on which species were produced during each run.

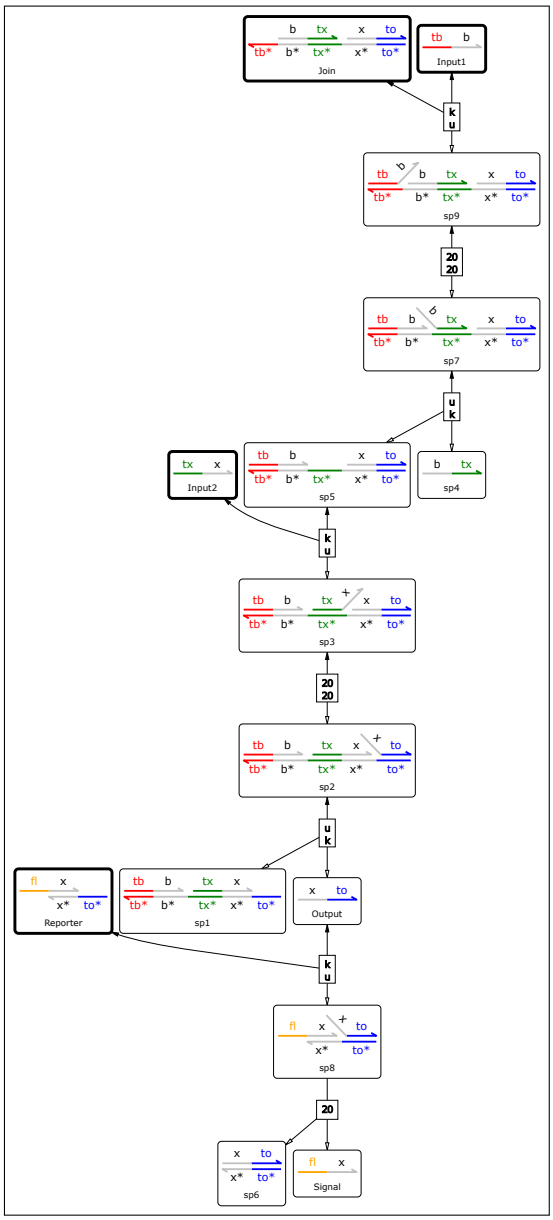
Using the JIT simulator offers significant practical advantages when handling large-scale systems such as those which include leak reactions. In many cases, simulating a system with leaks using the JIT simulator is comparable in speed to simulating that system without leaks using the stochastic simulator. Furthermore, the JIT simulator is essential when working with systems which have the potential to form DNA polymer molecules of unbounded size. In these cases the JIT simulator only compiles the subset of reactions which are reachable from a species that has been created during the simulation run, as opposed to the full (infinite) reaction graph. Thus the JIT simulator may be the only way to run DNA polymer programs without causing the compiler to loop forever. An example with JIT simulation is shown in Figure 23.

### 4.3.3 Leaks

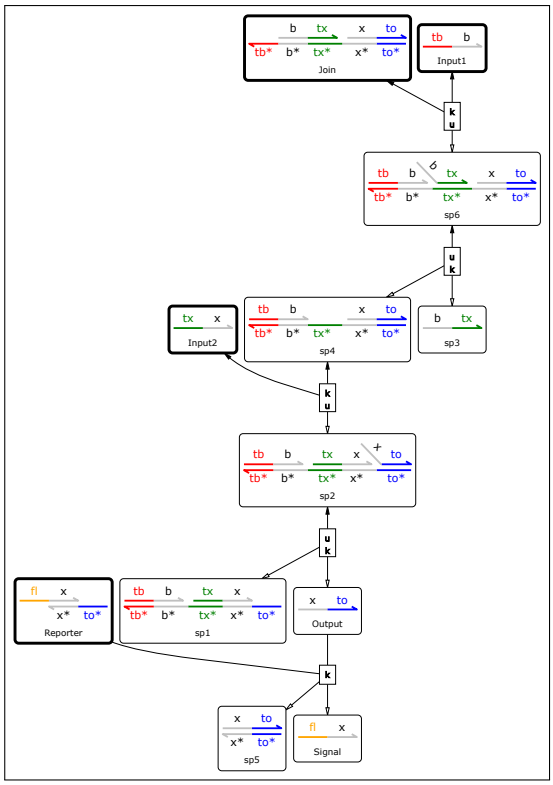
The `leak` directive instructs the Visual DSD compiler to automatically generate *leak reactions*, defined as DNA strand displacement reactions that occur in the absence of a toehold, albeit at very low rates. Generating leak reactions automatically produces a more realistic model of system behaviour, but increases the computational cost of both compilation and simulation. A leak reaction occurs when the nucleotides at one extremity of a bound strand spontaneously unbind, creating a short toehold that facilitates a strand displacement reaction. Consider the running example from Figure 16A. The `Input2` strand `<tx^ x>` can displace a bound `Output` strand `<x to^>` from the `Join` complex, even in absence of the `Input1` strand. This happens when one or two nucleotides at the 5′ end of the bound `Output` strand spontaneously unbind, creating a short toehold that allows the `x` domain of the `Input2` strand to displace the `Output`, where the default leak rate is $10^{-9}\text{nM}^{-1}\text{s}^{-1}$:



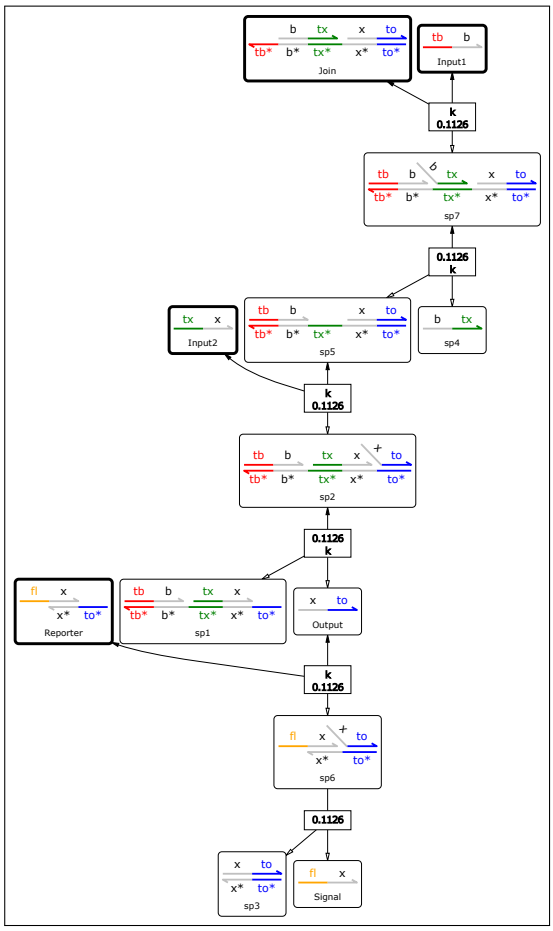

(a) Infinite compilation mode




(b) Detailed compilation mode

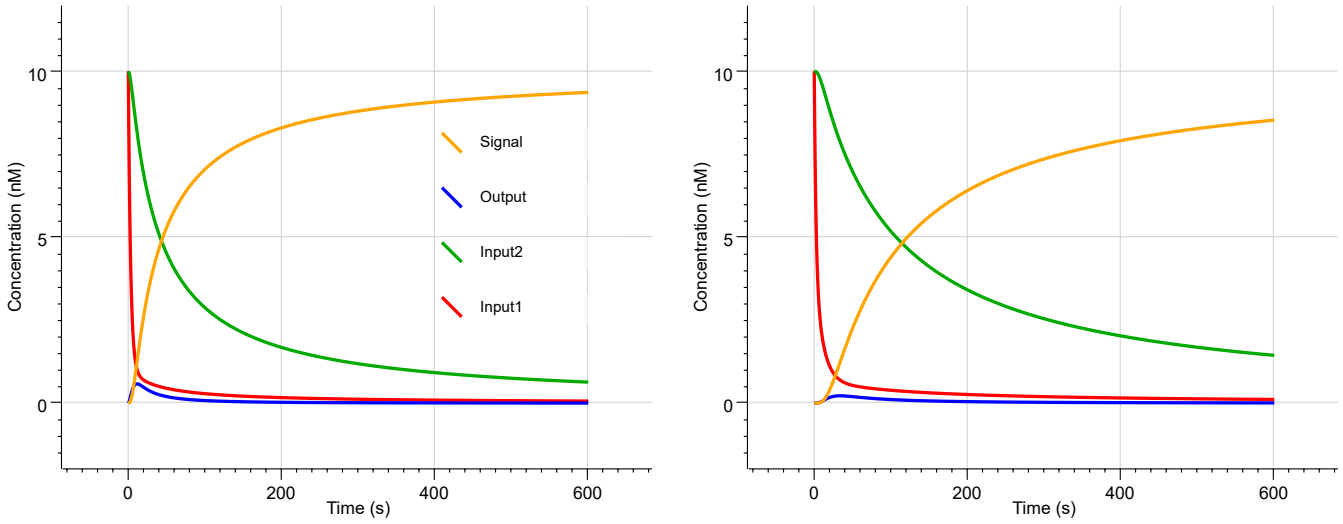



(c) Default compilation mode




(d) Finite compilation mode

**Figure 21: Chemical reaction networks for the different DSD compilation methods.**



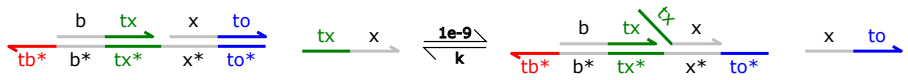

(a) Infinite compilation mode

(b) Default compilation mode

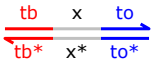
(c) Finite compilation mode

(d) Detailed compilation mode

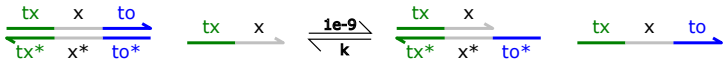
**Figure 22: Deterministic simulations for the different DSD compilation methods.**




The reverse reaction takes place at a much higher rate `k` since it is mediated by the toehold `to`. Note that a leak reaction cannot occur along a given domain if both ends of the domain are clamped in place by additional domains flanked on either side of the domain. For example, the `Input2` strand `<tx^ x>` cannot take part in a leak reaction with the complex `[tb^ x to^]` since the domain `x` is flanked by two toeholds on either side that clamp the domain in place.




Note that these flanking toeholds can themselves still fray and give rise to leak reactions, provided they also match the invading strand. For example, the `Input2` strand `<tx^ x>` can take part in a leak reaction with the complex `[tx^ x to^]`:

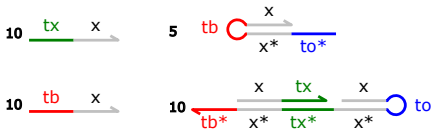



Enabling leak reactions can cause the number of reactions in the system to increase substantially. This is particularly apparent when the Finite or Detailed compilation modes are used. In these cases compilation can produce many thousands of reactions and take a long time to compute. To mitigate this, the system automatically filters out duplicate leak reactions with the same reactants and products as existing leak and non-leak reactions. These simplifications can reduce the number of reactions without significantly affecting system behaviour. However, in general

**A**

```
directive simulation {final=600}
directive simulator stochastic
directive jit
directive polymers
directive parameters [k=0.003;u=0.1]
directive compilation infinite
dom tb = {bind=k;unbind=u;colour="red"}
dom tx = {bind=k;unbind=u;colour="green"}
dom to = {bind=k;unbind=u;colour="blue"}
def Input1() = <tb^ x>
def Input2() = <tx^ x>
def Join() = {tb^*}[x tx^]:[x]{to^>
def Reporter() = <tb^}[x]{to^*}
( 10 Input1()
| 10 Input2()
| 10 Join()
| 5 Reporter()
)
```

**B**




**C**

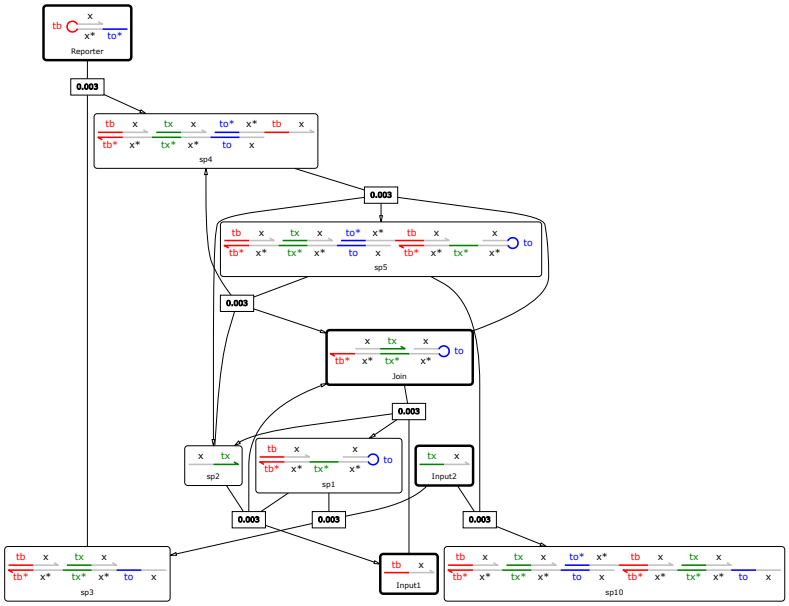



**Figure 23: Example with just-in-time simulation.**

it is still possible to derived large numbers of leak reactions from a seemingly small system. Note that Visual DSD does not currently support leak reactions which create DNA polymers by binding two gate molecules together. An example with leak reactions is shown in Figure 24. In many cases leak reactions have no noticeable effect on system behaviour, though this is difficult to determine a priori. A check can be performed by enabling leaks and, if no noticeable effects are observed, disabling leaks for subsequent analysis.

# References


[1] Andrew Phillips and Luca Cardelli. A programming language for composable DNA circuits. *J. R. Soc. Interface*, 6(Suppl_4):S419–S436, aug 2009.

[2] Matthew R. Lakin and Andrew Phillips. Modelling, Simulating and Verifying Turing-Powerful Strand Displacement Systems. In Luca Cardelli and William Shih, editors, *DNA Comput. Mol. Program.*, volume 6937 of *Lecture Notes in Computer Science*, pages 130–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[3] Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Localized Hybridization Circuits. In *DNA Comput. Mol. Program.*, volume 6937 LNCS, pages 64–83. 2011.

[4] Matthew R Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, nov 2011.

[5] Matthew R. Lakin, Loïc Paulevé, and Andrew Phillips. Stochastic simulation of multiple process calculi for biology. *Theor. Comput. Sci.*, 431:181–206, may 2012.

[6] Matthew R Lakin, Simon Youssef, Luca Cardelli, and Andrew Phillips. Abstractions for DNA circuit design. *J. R. Soc. Interface*, 9(68):470–486, mar 2012.

[7] Boyan Yordanov, Christoph M Wintersteiger, Youssef Hamadi, Andrew Phillips, and Hillel Kugler. Functional Analysis of Large-Scale DNA Strand Displacement Circuits. In David Soloveichik and Bernard Yurke, editors, *DNA Comput. Mol. Program.*, volume 8141 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2013.

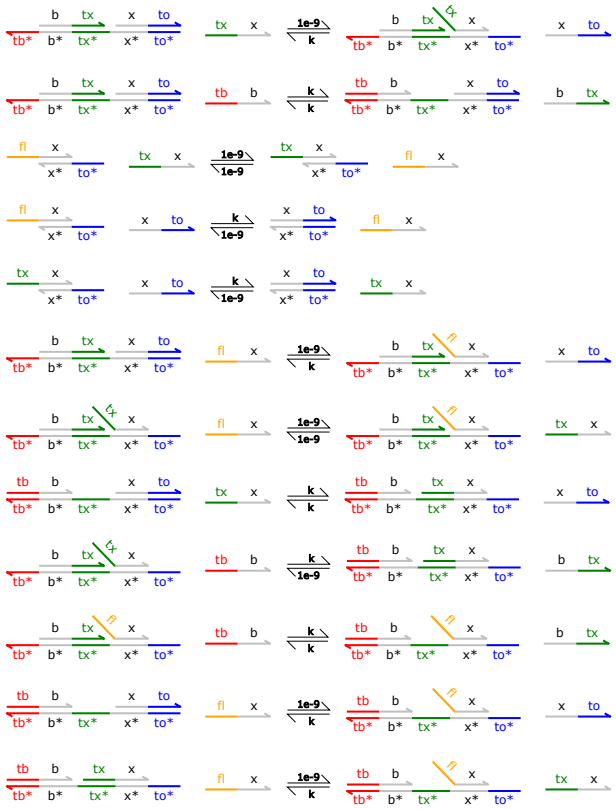
[8] Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nat. Nanotechnol.*, 8(10):755–762, sep 2013.

[9] Neil Dalchau, Georg Seelig, and Andrew Phillips. Computational Design of Reaction-Diffusion Patterns Using DNA-Based Chemical Reaction Networks. In *DNA Comput. Mol. Program.*, volume 8727, pages 84–99. 2014.

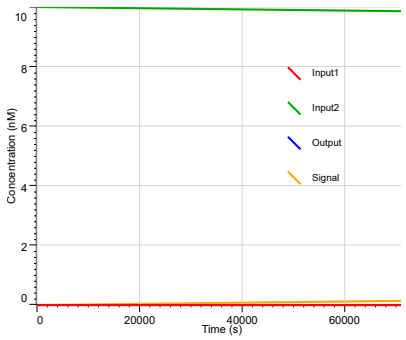
**A**

```
1  directive simulation {
2    final=72000;
3    plots=[Input1();Input2();Output();Signal()];
4  }
5  directive simulator deterministic
6  directive parameters [k=0.003;u=0.1]
7  directive compilation infinite
8  directive leaks
9  dom tb = {bind=k;unbind=u;colour="red"}
10 dom tx = {bind=k;unbind=u;colour="green"}
11 dom to = {bind=k;unbind=u;colour="blue"}
12 def Input1() = <tb^ b>
13 def Input2() = <tx^ x>
14 def Output() = <x to^>
15 def Join() = {tb^*}[b tx^]:[x to^]
16 def Reporter() = <fl^>[x]{to^*}
17 def Signal() = <fl^ x>
18 ( 0 Input1()
19 | 10 Input2()
20 | 0 Output()
21 | 100 Join()
22 | 100 Reporter()
23 | 0 Signal()
24 )
```

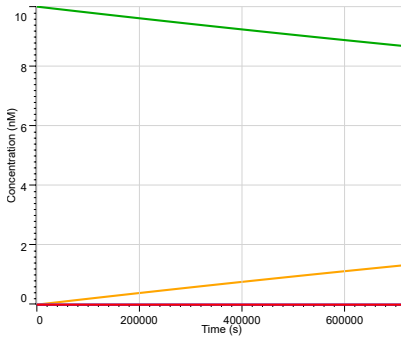
**B**



**C**




**D**




**E**

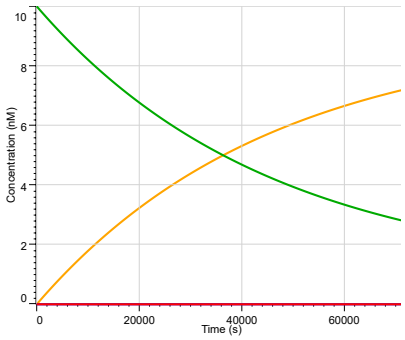



**Figure 24: Example with leaks.** **A** Program code, where differences from the running example are highlighted in grey. The automatic generation of leak reactions is enabled by writing `directive leaks` (line 8). To observe the effects of leaks, we increase the simulation time to 20 hours (line 2) and remove the presence of `Input1`. **B** CRN generated from (A), shown as a list of reactions from the Export/Reactions tab, including leak reactions. **C** Simulation results for (A). We observe that a small amount of `Signal` is produced after 20 hours, even in absence of `Input1`. This is achieved via a leak reaction between `Input2` and the `Reporter`, and also via a leak reaction between `Input2` and the `Join` complex that produces `Output`, which in turn produces `Signal` via a toehold mediated reaction with the `Reporter`. **D** Simulation results for (A) but for 200 hours, with `final = 720000`. We observe more substantial production of `Signal` but only minimal levels of `Output`, due to its slow production via a leak reaction and its much faster consumption via a toehold mediated reaction. **E** Simulation results for (A) but with the leak rate increased by two orders of magnitude. This is achieved by writing `directive leak 1e-7`, which allows a custom rate to be set for all leaks. Custom rates can also be set for individual leak reactions using user-defined reactions (Figure 18).


[10] Matthew R Lakin, Rasmus Petersen, Kathryn E Gray, and Andrew Phillips. Abstract Modelling of Tethered DNA Circuits. In *DNA Comput. Mol. Program.*, pages 132–147. 2014.

[11] Boyan Yordanov, Jongmin Kim, Rasmus L Petersen, Angelina Shudy, Vishwesh V Kulkarni, and Andrew Phillips. Computational Design of Nucleic Acid Feedback Control Circuits. *ACS Synth. Biol.*, 3(8):600–616, aug 2014.

[12] Neil Dalchau, Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Probabilistic Analysis of Localized DNA Hybridization Circuits. *ACS Synth. Biol.*, 4(8):898–913, aug 2015.

[13] Gourab Chatterjee, Neil Dalchau, Richard A Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nat. Nanotechnol.*, 12(9):920–927, jul 2017.

[14] Carlo Spaccasassi, Matthew R Lakin, and Andrew Phillips. A Logic Programming Language for Computational Nucleic Acid Devices. *ACS Synthetic Biology*, page acssynbio.8b00229, dec 2018.

[15] David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nat. Chem.*, 3(2):103–113, feb 2011.

[16] B Yurke, a J Turberfield, a P Mills, F C Simmel, and J L Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406(August):605–608, 2000.

[17] David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proc. Natl. Acad. Sci.*, 107(12):5393–5398, mar 2010.

[18] L. Qian and E. Winfree. Scaling Up Digital Circuit Computation with DNA Strand Displacement Cascades. *Science (80-. ).*, 332(6034):1196–1201, jun 2011.

[19] Nicolas Le Novère, Michael Hucka, Stefan Hoops, Sarah Keating, Sven Sahle, and Darren Wilkinson. Systems Biology Markup Language (SBML) Level 2: Structures and Facilities for Model Definitions. *Nat. Preced.*, 2008.

[20] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, 2005.

[21] Luca Cardelli. On process rate semantics. *Theor. Comput. Sci.*, 391(3):190–215, 2008.

[22] Christian P Robert and George Casella. *Monte Carlo Statistical Methods*, volume 95. 2004.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science (80-. ).*, 220(4598):671–680, 1983.

[24] Ranjan K. Mallik. The inverse of a tridiagonal matrix. *Linear Algebra Appl.*, 325(1-3):109–139, 2001.

[25] D W Peaceman and H H Rachford. The Numerical Solution of Parabolic and Elliptic Differential Equations. *Source J. Soc. Ind. Appl. Math.*, 3(1):28–41, 1955.

# A Visual CRN Language Summary

This appendix summarises all of the syntax definitions for the Visual CRN language that are present in the main text.

## A.1 Syntax Conventions

We first summarise the main syntax conventions of the Visual CRN language. Terminal symbols are written in `teletype` font and non-terminal symbols are written in *italics*. Comments begin with `(*` and end with `*)` and may be nested. Single line comments are written using `//` at the start of the comment. We use a standard syntax for *Records*, *Lists* and *Maps*, and assume that $N \geq 0$ unless stated otherwise:

**Records** are written as $\{Name_1 = Field_1; \ldots; Name_N = Field_N\}$, where $N \geq 1$. We assume that each *Name* in the record is unique and that each *Field* can potentially have a different type. We allow a *Field* to be omitted, in which case a default value is used.

**Lists** are written as $List\langle Value\rangle$, which is short for the syntax $[Value_1; \ldots; Value_N]$. We assume that every *Value* is of the same type.

**Maps** are written as $Map\langle Key, Value\rangle$, which is short for the syntax $[Key_1 = Value_1; \ldots; Key_N = Value_N]$. We assume that each *Key* in the map is unique and that every *Value* is of the same type.

We use the following syntax for *Integers*, *Names*, *Strings*, *Floats* and *Keywords*:

**Integers** are non-empty sequences of digits, where a digit denotes a single character in the range 0-9,

**Names** are sequences of characters, where the first character must either be a letter, which is a character in the range A-Z or a-z, or the underscore character (_). This is followed by a possibly-empty sequence of characters which may be letters, digits, underscores or apostrophes (').

**Strings** are possibly-empty sequences of characters enclosed by quotation marks (`"`). Any quotation marks appearing within the string must be escaped by a preceding backslash (\).

**Floats** can be written in three different ways: (i) One or more digits followed by a decimal point (.), followed by zero or more digits. For example 3.141. (ii) One or more digits followed by an uppercase E or lowercase e, followed by a plus (+) or minus (-) sign, followed by one or more digits. For example 3e-5. (iii) One or more digits followed by a decimal point, followed by zero or more digits, followed by an uppercase E or lowercase e, followed by a plus or minus sign, followed by one or more digits. For example 1.4324e+2.

**Keywords** are reserved words that cannot be used as a *Name*. The list of keywords is as follows: `constant directive def dom else false float_of_int if init int_of_float log new not prod rxn sum tether then time true`

## A.2 CRN Programs

| | |
|---|---|
| $Program ::= Settings\langle Species\rangle\ Modules\ Process$ | CRN Program |
| $Settings\langle S\rangle ::=$ `directive` $Setting\langle S\rangle_1 \ldots$ `directive` $Setting\langle S\rangle_N$ | Multiple settings |
| $Modules ::=$ `module` $Module_1 \ldots$ `module` $Module_N$ | Multiple modules |
| $Module ::= Name(Name_1, \ldots, Name_N) = \{Process\}$ | Module definition |
| $Species ::= Name$ | CRN species |
| $Process ::=$ | |
| $\mid Initial\langle Species\rangle$ | Initial condition |
| $\mid Reaction\langle Species\rangle$ | Reaction |
| $\mid Name(Value_1, \ldots, Value_N)$ | Module instance |
| $\mid Process_1 \mid \ldots \mid Process_N$ | Multiple parallel processes |

## A.3 Initial Conditions

| | |
|---|---|
| $Initial\langle S\rangle ::=$ | |
| $\mid Value\ S$ | Initial species population |
| $\mid Value_1\ S\ @Value_2$ | Delayed species population |
| $\mid$ `constant` $Value\ S$ | Constant species population |

| $S$ = *Attributes* | Species with attributes

| *Attributes* ::= { | Attributes record
|---|---|
| `value` = *Value*; | Species value (1) |
| `time` = *Value*; | Species delay (0) |
| `constant` = *Boolean*; | Species is constant (`false`) |
| `spatial` = *Placement*; | Species placement |
| } | |

## A.4 Reactions

| *Reaction*⟨$S$⟩ ::= | |
|---|---|
| \| *Reactants*⟨$S$⟩ `->` {*Value*} *Multiset*⟨$S$⟩ | Mass-action reaction |
| \| *Reactants*⟨$S$⟩ `<->` {*Value*} {*Value*} *Multiset*⟨$S$⟩ | Reversible mass-action reaction |
| \| *Reactants*⟨$S$⟩ `->` [*Functional*⟨$S$⟩] *Multiset*⟨$S$⟩ | Functional rate reaction |
| \| *Reactants*⟨$S$⟩ `<->` [*Functional*⟨$S$⟩] [*Functional*⟨$S$⟩] *Multiset*⟨$S$⟩ | Reversible functional rate reaction |

| *Reactants*⟨$S$⟩ ::= | |
|---|---|
| \| *Multiset*⟨$S$⟩ | Multiset of reactants |
| \| *Multiset*⟨$S$⟩∼*Multiset*⟨$S$⟩ | Multisets of catalysts and reactants |

| *Multiset*⟨$S$⟩ ::= $Value_1\ S_1 + \cdots + Value_N\ S_N$ | Multiset of species |
|---|---|

## A.5 Expressions

| *Value* ::= *Expression*⟨*Name*⟩ | Expression over parameters |
|---|---|
| *Functional*⟨$S$⟩ ::= *Expression*⟨*Key*⟨$S$⟩⟩ | Expression over species and parameters |

| *Key*⟨$S$⟩ ::= | |
|---|---|
| \| [$S$] | Species |
| \| [`time`] | Simulation time |
| \| [*Name*] | Functional rate |
| \| *Name* | Parameter name |

| *Expression*⟨$T$⟩ ::= | |
|---|---|
| \| $T$ | Expression variable |
| \| *Float* | Floating point number |
| \| *Expression*⟨$T$⟩ + *Expression*⟨$T$⟩ | Addition |
| \| *Expression*⟨$T$⟩ − *Expression*⟨$T$⟩ | Subtraction |
| \| *Expression*⟨$T$⟩ ∗ *Expression*⟨$T$⟩ | Multiplication |
| \| *Expression*⟨$T$⟩ / *Expression*⟨$T$⟩ | Division |
| \| *Expression*⟨$T$⟩%*Expression*⟨$T$⟩ | Modulo |
| \| *Expression*⟨$T$⟩^*Expression*⟨$T$⟩ | Power |
| \| *Expression*⟨$T$⟩ ∗ ∗*Expression*⟨$T$⟩ | Power |
| \| `sum`(*List*⟨*Expression*⟨$T$⟩⟩) | Sum |
| \| `prod`(*List*⟨*Expression*⟨$T$⟩⟩) | Product |
| \| `log`(*Expression*⟨$T$⟩) | Logarithm |
| \| `if` *Condition*⟨$T$⟩ | Conditional |
| `then` *Expression*⟨$T$⟩ | |
| `else` *Expression*⟨$T$⟩ | |

| *Condition*⟨$T$⟩ ::= | |
|---|---|
| \| `true` | Logical True |
| \| `false` | Logical False |
| \| *Expression*⟨$T$⟩ <= *Expression*⟨$T$⟩ | Less than or equal |
| \| *Expression*⟨$T$⟩ < *Expression*⟨$T$⟩ | Less than |
| \| *Expression*⟨$T$⟩ = *Expression*⟨$T$⟩ | Equal |
| \| *Expression*⟨$T$⟩ > *Expression*⟨$T$⟩ | Greater than |
| \| *Expression*⟨$T$⟩ >= *Expression*⟨$T$⟩ | Greater than or equal |

| *Condition*⟨*T*⟩&&*Condition*⟨*T*⟩ | Logical AND |
|---|---|
| *Condition*⟨*T*⟩||*Condition*⟨*T*⟩ | Logical OR |
| `not` *Condition*⟨*T*⟩ | Negation |

## A.6 CRN Settings

*Setting*⟨*S*⟩ ::=

| | |
|---|---|
| `parameters` *Map*⟨*Name*, *Parameter*⟩ | Parameter definitions |
| `sweeps` *Map*⟨*Name*, *List*⟨*Assignment*⟩⟩ | Sweep definitions |
| `rates` *Map*⟨*Name*, *Functional*⟨*S*⟩⟩ | Rate expressions |
| `plot_settings` *Plotting* | Plot format settings |
| `simulation` *Simulation*⟨*S*⟩ | Simulation settings |
| `simulations` *Map*⟨*Name*, *Simulation*⟨*S*⟩⟩ | Settings for multiple simulations |
| `simulator` *Simulator* | Simulator method (`stochastic`) |
| `deterministic` *Deterministic* | Deterministic simulation settings |
| `stochastic` *Stochastic* | Stochastic simulation settings |
| `spatial` *Spatial*⟨*S*⟩ | Spatial simulation settings |
| `moments` *Moments*⟨*S*⟩ | Moment closure simulation settings |
| `inference` *Inference* | Inference settings |
| `data` *List*⟨*Name*⟩ | Data files for inference |
| `units` *Units* | Units settings |

### A.6.1 Parameters

*Parameter* ::=

| | |
|---|---|
| *Value* | Parameter value |
| *Value*, *Prior* | Parameter value and associated prior |

### A.6.2 Sweeps

*Assignment* ::=

| | |
|---|---|
| *Name* = *Value* | Assign a value to a parameter name |
| $(Name_1, \ldots, Name_N) = (Value_1, \ldots, Value_N)$ | Assign a set of values to a set of names |

### A.6.3 Plotting

*Plotting* ::= {

| | |
|---|---|
| `x_label` = *String*; | X axis plot label |
| `y_label` = *String*; | Y axis plot label |
| `title` = *String*; | Plot title |
| `label_font_size` = *Integer*; | Font size of labels |
| `tick_font_size` = *Integer*; | Font size of axis numbers |
| `x_ticks` = `List`⟨`Integer`⟩; | Specific X axis numbers |
| `y_ticks` = `List`⟨`Integer`⟩; | Specific Y axis numbers |

}

### A.6.4 Units

*Units* ::= {

| | |
|---|---|
| `time` = *Time*; | Time units (`s`) |
| `space` = *Space*; | Space units (`m`) |
| `concentration` = *Concentration*; | Concentration units (`nM`) |

}

*Time* ::=

| | |
|---|---|
| `h` | hours (3600s) |
| `min` | minutes (60s) |
| `s` | seconds (s) |
| `ms` | milliseconds ($10^{-3}$ s) |
| `us` | microseconds ($10^{-6}$ s) |
| `ns` | nanoseconds ($10^{-9}$ s) |

*Space* ::=

| | |
|---|---|
| `m` | metres (m) |
| `mm` | millimetres ($10^{-3}$ m) |
| `um` | micrometres ($10^{-6}$ m) |
| `nm` | nanometres ($10^{-9}$ m) |
| `pm` | picometres ($10^{-12}$ m) |
| `fm` | femtometres ($10^{-15}$ m) |

*Concentration* ::=

| | |
|---|---|
| `M` | molar (M) |
| `mM` | millimolar ($10^{-3}$ M) |
| `uM` | micromolar ($10^{-6}$ M) |
| `nM` | nanomolar ($10^{-9}$ M) |
| `pM` | picomolar ($10^{-12}$ M) |
| `fM` | femtomolar ($10^{-15}$ M) |
| `aM` | attomolar ($10^{-18}$ M) |
| `zM` | zeptomolar ($10^{-21}$ M) |
| `yM` | yoctomolar ($10^{-24}$ M) |

### A.6.5 Simulation

| | |
|---|---|
| *Simulation*⟨*S*⟩ ::= { | Simulation record |
| `initial` = *Float*; | Initial simulation time (0) |
| `final` = *Float*; | Final simulation time (1000) |
| `points` = *Integer*; | Number of simulation points to plot (1000) |
| `plots` = *Map*⟨*Plot*⟨*S*⟩⟩; | Species to plot. Plots all species if empty ([]) |
| `kinetics` = *Kinetics*; | Kinetic rate convention for homomultimer formation (`Contextual`) |
| `prune` = *Boolean*; | Remove unreachable reactions prior to simulation (`false`) |
| `multicore` = *Boolean*; | Use multiple CPU cores to run simulation, if present (`false`) |
| `data` = *List*⟨*Name*⟩; | List of Datasets to compare with simulation results ([]) |
| `sweeps` = *List*⟨*Name*⟩; | List of sweeps to use for simulation ([]) |
| } | |

*Plot*⟨*S*⟩ ::=

| | |
|---|---|
| *S* | Single species |
| *Functional*⟨*S*⟩ | Function of species, parameters or rates |

*Kinetics* ::=

| | |
|---|---|
| `Contextual` | Kinetic convention depends on simulator directive |
| `Stochastic` | Stochastic kinetic convention |
| `Deterministic` | Deterministic kinetic convention |

### A.6.6 Simulator

*Simulator* ::=

| | |
|---|---|
| `deterministic` | Deterministic simulation of ordinary differential equations (OSLO solvers) |
| `sundials` | Deterministic simulation of ordinary differential equations (SUNDIALs solvers) |
| `stochastic` | Stochastic simulation using the Gillespie Stochastic Simulation Algorithm (SSA) |
| `lna` | Mean and variance over time of a stochastic system (Linear Noise Approximation) |
| `cme` | Probability distribution over time of a stochastic system (Chemical Master Equation) |
| `pde` | Deterministic spatiotemporal simulation of reaction-diffusion equations |

### A.6.7 Deterministic

| | |
|---|---|
| *Deterministic* ::= { | Deterministic record |
| `stiff` = *Boolean*; | Enable stiff solver (`false`) |
| `abstolerance` = *Float*; | Relative tolerance (1E-5) |
| `reltolerance` = *Float*; | Absolute tolerance (1E-6) |
| } | |

### A.6.8 Stochastic

| | |
|---|---|
| *Stochastic* ::= { | Stochastic record |
| `scale` = *Float*; | Scale volume while fixing concentrations (1) |
| `seed` = *Option*⟨*Integer*⟩; | Random number generator seed (∅, uses random seed) |
| `steps` = *Option*⟨*Integer*⟩; | Number of simulation steps to take (∅, does not limit steps) |
| `trajectories` = *Integer*; | Number of simulation repeats, plotting mean and variance (1) |

}

### A.6.9 Spatial

| | |
|---|---|
| *Spatial*⟨*S*⟩ ::= { | Spatial record |
| `diffusibles` = *Map*⟨*S*, *Value*⟩; | Assigns diffusion rates to species ([]) |
| `default_diffusion` = *Float*; | Default diffusion rate for all species (0) |
| `dimensions` = *Integer*; | Number of spatial dimensions (1) |
| `random` = *Float*; | Relative random perturbation to initial conditions (0) |
| `xmax` = *Float*; | Maximum length of spatial domain (1) |
| `nx` = *Integer*; | Number of spatial grid points, including boundary (20) |
| `dt` = *Float*; | Time step for spatial simulations ($\frac{0.2dx^2}{maxdiffusibles}$) |
| `boundary` = *Boundary*; | Boundary conditions for spatial simulation (`Periodic`) |
| } | |

| | |
|---|---|
| *Bloundary* ::= | |
| \| *Periodic* | Periodic boundary conditions |
| \| *ZeroFlux* | Zero flux boundary conditions |

| | |
|---|---|
| *Placement* ::= { | Placement record |
| `centralcore` = *Core*; | Central core |
| `points` = `List`⟨`Point`⟩; | A list of points ([]) |
| `random` = *Float*; | Add random noise to initial population (0) |
| } | |

| | |
|---|---|
| *Core* ::= { | Core record |
| `width` = *Float*; | Diameter of central core, in relative units (0) |
| `inner` = *Float*; | Concentration inside central core (0) |
| `outer` = *Float*; | Concentration outside central core (0) |
| } | |

| | |
|---|---|
| *Point* ::= { | Point record |
| `x` = *Float*; | Relative $x$ position of point (0) |
| `y` = *Float*; | Relative $y$ position of point (0) |
| `width` = *Float*; | Diameter of point, in relative units (0) |
| `value` = *Float*; | Concentration of species inside point (0) |
| } | |

### A.6.10 Moments

| | |
|---|---|
| *Moment*⟨*S*⟩ ::= { | Moment record |
| `order` = *Integer*; | Moment closure order |
| `initial_minimum` = *Float*; | Initial minimal value for all species |
| `log_evaluation` = *Boolean*; | Enable log evaluation near zero populations |
| `plots` = *List*⟨*Monomial*⟨*S*⟩$_1$ ∗ ⋯ ∗ *Monomial*⟨*S*⟩$_N$⟩; | Moment closure plots, $N \geq 1$ |
| } | |

| | |
|---|---|
| *Monomial*⟨*S*⟩ ::= *S*^*Integer* | Moment monomial |

### A.6.11 Inference

| | |
|---|---|
| *Inference* ::= { | Inference record |
| `name` = *Name*; | Name of inference run (`default`) |
| `burnin` = *Integer*; | Number of discarded initial iterations (100) |
| `samples` = *Integer*; | Number of samples stored in posterior (100) |
| `thin` = *Integer*; | Ratio of posterior samples to discard (10) |
| `noise_model` = *Noise*; | Noise model (`constant`) |
| `prune` = *Boolean*; | Remove reactions involving species that are never produced (`false`) |

| | |
|---|---|
| `seed` = *Integer*; | Seed of random number generator (0) |
| `timer` = *Boolean*; | Collect timing statistics for simulations (`false`) |
| `partial` = *Boolean*; | Only run simulations affected by changed parameters (`false`) |
| } | |

*Noise* ::=

| | |
|---|---|
| \| `constant` | Error is constant |
| \| `proportional` | Error is proportional to measured value |

| | |
|---|---|
| *Prior* ::= { | Prior record |
| `interval` = *Interval*; | Scale used to vary the parameter |
| `distribution` = *Distribution*; | Prior distribution of the parameter |
| `variation` = *Variation*; | Method used to vary the parameter |
| } | |

*Interval* ::=

| | |
|---|---|
| \| *Real* | Vary the parameter on a linear scale |
| \| *Log* | Vary the parameter on a logarithmic scale |

*Distribution* ::=

| | |
|---|---|
| \| `Uniform`($Float_1$, $Float_2$) | Uniform distribution between $Float_1$ and $Float_2$ |
| \| `Normal`($Float_1$, $Float_2$) | Normal distribution, mean $Float_1$, standard deviation $Float_2$ |
| \| `TruncatedNormal` ($Float_1$, $Float_2$, $Float_3$, $Float_4$) | Truncated Normal distribution, mean $Float_1$, standard deviation $Float_2$, between $Float_3$ and $Float_4$ |

*Variation* ::=

| | |
|---|---|
| \| *Random* | Randomly vary parameter |
| \| *Fixed* | Keep parameter fixed |
| \| *Initial* | Use default value as initial estimate |
| \| *Multiple* | Use a unique parameter in each sweep instance |

# B Spatial Simulation Methods

Here we describe how numerical solutions are obtained to reaction-diffusion equations of the form:

$$\frac{\partial \boldsymbol{u}}{dt}(\boldsymbol{x},t) = \boldsymbol{f}(\boldsymbol{u}(\boldsymbol{x},t)) + D\nabla^2\boldsymbol{u}(\boldsymbol{x},t) \tag{10}$$

where $u(\boldsymbol{x},t)$ is a vector of concentrations at time $t$ and position $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{f}$ describes the fluxes of the reactions, and is equivalent to deterministic simulation. The diagonal matrix $D$ expresses the diffusion rates of the species in $\boldsymbol{u}$. The Crank-Nicolson method is used to provide numerical solutions, and currently supports 1d domains with periodic or zero-flux boundary conditions, and 2d domains with periodic boundaries.

## B.1 The 1d problem

The description of this method is based on the slides from Douglas Wilhelm Harder (University of Waterloo, Ontario), which are available from https://ece.uwaterloo.ca/~ne217/Laboratories/03/3.CrankNicolson.pptx.

The 1d version of (10) is given by

$$\frac{\partial u}{dt}(x,t) = f(u(x,t)) + D\frac{\partial^2 u}{\partial x^2}(x,t) \tag{11}$$

We consider the 1d problem on a domain $x \in [a,b]$.

The Crank-Nicolson method uses a combination of the explicit and implicit finite differences applied to a regular grid of $m+1$ gridpoints such that $u(a,t_k) = u_{0,k}$ and $u(b,t_k) = u_{n_x,k}$. The explicit scheme uses the following finite difference approximations

$$\frac{\partial u}{dt}(x_i,t_k) = \frac{u_{i,k+1} - u_{i,k}}{\Delta t} \tag{12a}$$

$$\frac{\partial^2 u}{\partial x^2}(x_i,t_k) = \frac{u_{i-1,k} - 2u_{i,k} + u_{i+1,k}}{h^2} \tag{12b}$$

This leads to a recurrence relation

$$u_{i,k+1} = u_{i,k} + \Delta t.f(u_{i,k}) + \frac{D.\Delta t}{h^2}\left(u_{i-1,k} - 2u_{i,k} + u_{i+1,k}\right) \tag{13}$$

The implicit scheme attempts to match grid-points at the next time interval rather than the current on. Accordingly, the finite difference approximations used are

$$\frac{\partial u}{dt}(x_i,t_{k+1}) = \frac{u_{i,k+1} - u_{i,k}}{\Delta t} \tag{14a}$$

$$\frac{\partial^2 u}{\partial x^2}(x_i,t_{k+1}) = \frac{u_{i-1,k} - 2u_{i,k} + u_{i+1,k}}{h^2} \tag{14b}$$

This leads to a recurrence relation

$$u_{i,k+1} = u_{i,k} + \Delta t.f(u_{i,k}) + \frac{D.\Delta t}{h^2}\left(u_{i-1,k+1} - 2u_{i,k+1} + u_{i+1,k+1}\right) \tag{15}$$

Adding together the explicit and implicit schemes leads to a centred time-evolution. In principle, it is possible to add together the schemes in a bias-weighted way, for instance doing $\theta *$ implicit $+ (1-\theta) *$ explicit, but we omit details here for simplicity. A even weighting corresponds summing equations (13) and (15), which after bringing time $k+1$ terms to the left and $k$ terms to the right, gives

$$-ru_{i-1,k+1} + 2(r+1)u_{i,k+1} - ru_{i+1,k+1} = 2u_{i,k} + 2\Delta t.f(u_{i,k}) + r(u_{i-1,k} - 2u_{i,k} + u_{i+1,k}) \tag{16}$$

where

$$r = \frac{D.\Delta t}{h^2} \tag{17}$$

is a quantity we now refer to as the *flux constant*.

To complete the problem setup, we must consider the boundary conditions that will be used.

### B.1.1 Zero-flux boundary conditions (1st order)

For a 1d domain $x \in [a, b]$, zero-flux (or insulated) boundaries are defined by

$$\frac{\partial u}{\partial x}(a, t_{k+1}) = 0 \tag{18a}$$
$$\frac{\partial u}{\partial x}(b, t_{k+1}) = 0 \tag{18b}$$

Using a 1st order approximation, we assign

$$\frac{\partial u}{\partial x}(a, t_{k+1}) \approx \frac{u(a,t) - u(a+h,t)}{h} \tag{19a}$$
$$\frac{\partial u}{\partial x}(b, t_{k+1}) \approx \frac{u(b,t) - u(b-h,t)}{h} \tag{19b}$$

which leads to the recurrence relations

$$u_{0,k+1} - u_{-1,k+1} = 0 \tag{20a}$$
$$u_{n_x,k+1} - u_{n_x+1,k+1} = 0 \tag{20b}$$

This enables us to account for the points $u_{-1,k+1}$ and $u_{n_x+1,k+1}$ that lie outside of the grid.

With these corrections, we can define a linear system to be solved over the $u_{i,k}$, as

$$A_z \boldsymbol{u}_{k+1} = B_z \boldsymbol{u}_k + 2f(\boldsymbol{u}_k) \tag{21}$$

where

$$A_z = \begin{pmatrix} 2+r & -r & & & \\ -r & 2(1+r) & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 2(1+r) & -r \\ & & & -r & 2+r \end{pmatrix}, \quad B_z = \begin{pmatrix} 2-r & r & & & \\ r & 2(1-r) & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 2(1-r) & r \\ & & & r & 2-r \end{pmatrix}$$

As $A_z$ is a tridiagonal matrix, we can use the Thomas' tridiagonal algorithm [24] to solve for $\boldsymbol{u}_{k+1}$ efficiently.

### B.1.2 Zero-flux boundary conditions (2nd order)

Using a 2nd order approximation, we obtain

$$\frac{\partial u}{\partial x}(a, t_{k+1}) \approx \frac{-3u(a,t) + 4u(a+h,t) - u(a+2h,t)}{2h} \tag{22a}$$
$$\frac{\partial u}{\partial x}(b, t_{k+1}) \approx \frac{u(b-2h,t) - 4u(b-h,t) + 3u(b,t)}{2h} \tag{22b}$$

which leads to the recurrence relations

$$-u_{2,k+1} + 4u_{1,k+1} - 3u_{0,k+1} = 0 \tag{23a}$$
$$u_{n_x,k+1} - 4u_{n_x-1,k+1} + u_{n_x-2,k+1} = 0 \tag{23b}$$

These conditions can be used to eliminate $u_{0,k+1}$ and $u_{n_x,k+1}$ from a system of $n_x - 1$ equations for the internal points, which becomes:

$$A_z u_{1:n_x-1,k+1} = B_z u_{0:n_x,k} + 2f(u_{1:n_x,k}) \tag{24}$$

where

$$A_z = \begin{pmatrix} 2+\frac{2}{3}r & -\frac{2}{3}r & & & \\ -r & 2(1+r) & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 2(1+r) & -r \\ & & & -\frac{2}{3}r & 2+\frac{2}{3}r \end{pmatrix}, \quad B_z = \begin{pmatrix} 2-r & r & & & \\ r & 2(1-r) & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 2(1-r) & r \\ & & & r & 2-r \end{pmatrix}$$

As before, we can use Thomas' tridiagonal algorithm to solve for $\boldsymbol{u}_{1:n_x-1,k+1}$, but this time fill in the end-points with corrections according to (23).

### B.1.3 Periodic boundary conditions

For periodic boundaries in 1d, the idea is to equate $u_{0,k}$ to $u_{n_x,k}$. Starting from equation (16), we seek to form a matrix equation, analogous to (21). Instead, we seek to wrap around a solution for the grid-points $0, \ldots, n_x - 1$. Accordingly, we obtain

$$A_p u_{0:n_x-1,k+1} = B_p u_{0:n_x-1,k} + 2f(u_{0:n_x-1,k}) \tag{25}$$

where

$$A_p = \begin{pmatrix} 2(1+r) & -r & & & -r \\ -r & 2(1+r) & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 2(1+r) & -r \\ -r & & & -r & 2(1+r) \end{pmatrix}, \quad B_p = \begin{pmatrix} 2(1-r) & r & & & r \\ r & 2(1-r) & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 2(1-r) & r \\ r & & & r & 2(1-r) \end{pmatrix}$$

Unfortunately, as $A_p$ is not completely tridiagonal, and so we must solve the matrix equation using a more general technique. As $A_p$ is unchanged over time, an efficient (yet numerically unstable) method is to simply calculate $A_p^{-1}$ upfront, then use matrix multiplication at each time-step to evaluate

$$u_{0:n_x-1,k+1} = A_p^{-1}\left(B_p u_{0:n_x-1,k} + 2f(u_{0:n_x-1,k})\right)$$

To complete the solution for $u$ at time $t_{k+1}$, we simply copy the value at $x = a$ into the storage for $x = b$, ensuring periodicity.

An alternative approach is to use the Sherman-Morrison formula (see Section B.3.2).

## B.2 The 2d problem

To solve the 2d problem, we make use of alternating direction implicit (ADI) methods, as described by Peaceman and Rachford in 1955 [25]. A nice discussion on the use of ADI methods is provided by Taras Lakoba (University of Vermont) in his teaching materials, which are available at the time of writing from http://www.cems.uvm.edu/~tlakoba/math337/notes_15.pdf.

The 2d version of (10) is given by

$$\frac{\partial u}{dt}(x,y,t) = f(u(x,y,t)) + D\left(\frac{\partial^2 u}{\partial x^2}(x,y,t) + \frac{\partial^2 u}{\partial y^2}(x,y,t)\right) \tag{26}$$

We consider the 2d problem on a domain $(x,y) \in [a,b] \times [a,b]$. Accordingly, we extend our previous notation such that $u(a+ih, a+jh, t_k) = u_{i,j,k}$.

### B.2.1 Simple approaches

We begin by introducing the explicit update scheme for the 2d problem, which gives

$$\frac{u_{i,j,k+1} - u_{i,j,k}}{\Delta t} = f(u_{i,j,k}) + D\left(\frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2}\right) \tag{27}$$

which can immediately be simplified by imposing $\Delta x = \Delta y = h$ and defining

$$r = \frac{D\Delta t}{h^2} \tag{28}$$

as in the 1d case. Accordingly, we obtain the recurrence relation

$$u_{i,j,k+1} = f(u_{i,j,k})\Delta t + u_{i,j,k} + r\left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} - 4u_{i,j,k}\right) \tag{29}$$

The implicit scheme is similar, but the space derivatives are considered at the next time-step, rather than the current one. As such,

$$\frac{u_{i,j,k+1} - u_{i,j,k}}{\Delta t} = f(u_{i,j,k}) + D\left(\frac{u_{i-1,j,k+1} - 2u_{i,j,k+1} + u_{i+1,j,k+1}}{\Delta x^2} + \frac{u_{i,j-1,k+1} - 2u_{i,j,k+1} + u_{i,j+1,k+1}}{\Delta y^2}\right) \tag{30}$$

Rearranging the terms gives a recurrence relation

$$u_{i-1,j,k+1} + u_{i+1,j,k+1} + u_{i,j-1,k+1} + u_{i,j+1,k+1} - (4 + \frac{1}{r})u_{i,j,k+1} = -\frac{1}{r}\left(u_{i,j,k} + f(u_{i,j,k})\Delta t\right) \tag{31}$$

### B.2.2 Alternating direction implicit (ADI) method

In both the explicit and implicit space derivatives, and also in a similar Crank-Nicolson scheme, one obtains a recurrence relation involving a *block* tridiagonal system of equations, for which no efficient solution exists [25]. However, the ADI methods circumvent this problem, producing an approximate tridiagonal system. The ADI method uses two difference equations, essentially performing an implicit update in the x-direction in the first half of a step, and an update in the y-direction in the second half of a step. Therefore, adapting equations (29) and (31) derived above, we have

$$u_{i,j,2k+1} = u_{i,j,2k} + \frac{\Delta t}{2}.f(u_{i,j,2k}) + \hat{r}.\left(u_{i-1,j,2k+1} - 2u_{i,j,2k+1} + u_{i+1,j,2k+1} + u_{i,j-1,2k} - 2u_{i,j,2k} + u_{i,j+1,2k}\right) \tag{32a}$$

$$u_{i,j,2k+2} = u_{i,j,2k+1} + \frac{\Delta t}{2}.f(u_{i,j,2k+1}) + \hat{r}.\left(u_{i-1,j,2k+1} - 2u_{i,j,2k+1} + u_{i+1,j,2k+1} + u_{i,j-1,2k+2} - 2u_{i,j,2k+2} + u_{i,j+1,2k+2}\right) \tag{32b}$$

where an alternative flux constant $r_2$ is defined as

$$\hat{r} = \frac{D\Delta t}{2h^2} = \frac{r}{2} \tag{33}$$

Rearranging to separate $2k$, $2k+1$ and $2k+2$ terms, we obtain

$$-\hat{r}.\left(u_{i-1,j,2k+1} - 2u_{i,j,2k+1} + u_{i+1,j,2k+1}\right) + u_{i,j,2k+1} = u_{i,j,2k} + \frac{\Delta t}{2}.f(u_{i,j,2k}) + \hat{r}.\left(u_{i,j-1,2k} - 2u_{i,j,2k} + u_{i,j+1,2k}\right) \tag{34a}$$

$$-\hat{r}.\left(u_{i,j-1,2k+2} - 2u_{i,j,2k+2} + u_{i,j+1,2k+2}\right) + u_{i,j,2k+2} = u_{i,j,2k+1} + \frac{\Delta t}{2}.f(u_{i,j,2k+1}) + \hat{r}.\left(u_{i-1,j,2k+1} - 2u_{i,j,2k+1} + u_{i+1,j,2k+1}\right) \tag{34b}$$

Therefore, we can solve these equations by first iterating over $j = 1 \dots n_y - 1$, solving

$$A_p U^*_{:,j} = \frac{\Delta t}{2} f(u_{0:n_x,j,2k}) + \hat{r}U_{:,j-1} + (1-2\hat{r})U_{:,j} + \hat{r}U_{:,j+1} \tag{35}$$

where $U_{:,j}$ is the $j^{th}$ column of $u_{i,j,2k}$ and $U^*_{:,j}$ is the $j^{th}$ column of $u_{i,j,2k+1}$. For $j = 0$ and $j = n_y$, we do something that depends on the boundary conditions.

In the second step, we iterate over $i = 1 \dots n_x - 1$, solving

$$A_p U_{i,:} = \frac{\Delta t}{2} f(u_{i,0:n_y,2k}) + \hat{r}U^*_{i-1,:} + (1-2\hat{r})U^*_{i,:} + \hat{r}U^*_{i+1,:} \tag{36}$$

at which point $U_{i,:}$ describes the $i^{th}$ row of the solution having gone through both steps. Again, we must handle $i = 0$ and $i = n_x$ as special cases.

Furthermore, the precise form of $A_p$ depends on the boundary conditions. As for the 1d case, we can collect the coefficients of $U$ and $U^*$ of the right-hand sides into a matrix $B_p$. Then, we can compactly represent the equations to solve as

$$A_p U^* = \frac{\Delta t}{2} f(u_{0:n_x,j,2k}) + B_p U \tag{37a}$$

$$A_p U = \frac{\Delta t}{2} f(u_{i,0:n_y,2k}) + B_p U^* \tag{37b}$$

### B.2.3 Neumann boundary conditions

To apply zero-flux boundary conditions to equations (37), we can adopt a first order approximation, equivalent to that done in 1d (Section B.1.1). This gives rise to the relationships

$$u_{0,j,k+1} - u_{-1,j,k+1} = 0 \quad (\forall j) \tag{38a}$$

$$u_{n_x,j,k+1} - u_{n_x+1,j,k+1} = 0 \quad (\forall j) \tag{38b}$$

$$u_{i,0,k+1} - u_{i,-1,k+1} = 0 \quad (\forall i) \tag{38c}$$

$$u_{i,n_y,k+1} - u_{i,n_x+1,k+1} = 0 \quad (\forall i) \tag{38d}$$

This leads us to define the triadiagonal matrices

$$A_p = \begin{pmatrix} 1+r & -r & & & \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+r \end{pmatrix}, B_p = \begin{pmatrix} 1-r & r & & & \\ r & 1-2r & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 1-2r & r \\ & & & r & 1-r \end{pmatrix}$$

As the matrices are tridiagonal, we use the Thomas algorithm (see Section B.3.1) to solve $A_p U = x$ at each step.

### B.2.4 Periodic boundary conditions

To apply periodic boundary conditions to equations (37), we wrap around the edges of domains in each direction. So, we replace the terms at $i = n_x$ with those at $i = 0$, and similarly for $i = -1$ with $i = n_x - 1$, and then the equivalent replacements for $j = n_y$ and $j = -1$. This leads us to define the cyclic tridiagonal matrix

$$A_p = \begin{pmatrix} 1+2r & -r & & & -r \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ -r & & & -r & 1+2r \end{pmatrix}, B_p = \begin{pmatrix} 1-2r & r & & & r \\ r & 1-2r & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 1-2r & r \\ r & & & r & 1-2r \end{pmatrix}$$

As the matrices are cyclic tridiagonal, we use the Sherman-Morrison algorithm (see Section B.3.2) to solve $A_p U = x$ at each step.

## B.3 Useful algorithms

### B.3.1 Thomas' tridiagonal algorithm

### B.3.2 Sherman-Morrison Formula

Consider the problem

$$Ax = u \tag{39}$$

where $A$ is a cyclic tridiagonal matrix

$$A = \begin{pmatrix} b_1 & c_1 & 0 & \dots & 0 & 0 & a_N \\ a_1 & b_2 & c_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{N-2} & b_{N-1} & c_{N-1} \\ c_N & 0 & 0 & \dots & 0 & a_{N-1} & b_N \end{pmatrix}$$

The idea is to separate $A$ into the sum of a tridiagonal matrix and an outer product of two vectors according to

$$A = Q + v \otimes w$$

where

$$v = \begin{pmatrix} \beta \\ 0 \\ \vdots \\ 0 \\ c_N \end{pmatrix}, \quad w = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ a_N/\beta \end{pmatrix}$$

Accordingly, the bottom-left and top-right corners of $Q$ become 0. By assigning $\beta = -b_1$, the top-left and bottom-right corners become

$$A_{11} = 2b_1, \quad A_{NN} = b_N + a_N c_N / b_1$$

Finally, we solve (39) by solving the two equations

$$Qy = u, \quad Qz = v$$

using the Thomas algorithm ($Q$ tridiagonal). Accordingly,

$$x = y - \left( \frac{w \cdot y}{1 + (w \cdot z)} \right) z \tag{40}$$