Visual DSD Tutorial

Background

Visual DSD is a programming language for designing, simulating and analysing systems of DNA molecules that interact via strand displacement. The tool is intended for use by researchers and students with an understanding of basic concepts in DNA computing.

Preliminaries: Installing Visual DSD

1. Navigate to http://research.microsoft.com/dna and click on the "Web Simulator" link.
2. Visual DSD requires Silverlight 4 to be installed on your machine. If this is not the case, you should receive a notification in your browser together with installation instructions. Silverlight 4 is available for Windows and Mac as a plugin for most major web browsers.
3. In the top-right corner, click on the "Install" button. This will install the software to your local machine for offline use. You will be presented with options to create shortcuts on the Start menu and the Desktop. If you have already installed the tool, the "Install" button will read "Update" and will check if a newer version of the tool is available to download.

I: Basic compilation and simulation

1. <u>Load the "Catalytic" example from the "Examples:" drop-down menu.</u> The program code will appear in the "Code" tab on the left-hand side. This system implements a catalytic gate using DNA strand displacement, as described by Zhang et al. (Science 318:1121-1125, 2007). The goal of the catalytic gate is to release large quantities of output if a particular catalyst species is present.
2. The code describes a collection of DNA species which interact to produce the behaviour of a catalytic gate. <u>Use the "Compile" button to calculate all possible interactions between these species and their products.</u> This produces output in the various sub-tabs of the "Compilation" tab on the right-hand side. In particular, observe the graphical representation of the chemical reactions in the "Graph" sub-tab. The compilation also produces a graphical representation of the input species in the "Input" tab on the left hand side. <u>Use the top four options in the "View" drop-down menu to modify the graphical presentation. The "Complement" view is selected by default, which illustrates the binding of DNA strands along complementary regions of their DNA sequence.</u>
3. Having compiled the reactions we can now simulate them. <u>Switch to the "Simulation" tab on the right-hand side and select the "Plot" sub-tab. Use the "Simulate" button to run a stochastic simulation of the catalytic gate system.</u> This produces output in the sub-tabs of the "Simulation" tab, including the time course "Plot", a data "Table" and visualisations of the "Initial state" and "Last state" of the system. In the time course, note that the populations of the <6 3^ 4> and <1 2> strands increase over time as they are produced by the catalytic cycle of the gate in response to the presence of the <4 5^> catalyst species.
4. The first line of the program code controls the duration of the simulation run and the number of population samples to take during that time. <u>Modify the code to run for 2000</u>

time units, with a sample taken every 4 time units, then re-compile the code and re-run the stochastic simulation.

5. Visual DSD supports multiple levels of abstraction for compiling the interactions between DNA species. These are selected from the “Compilation:” drop-down menu, where “Infinite” is the most abstract semantics and “Detailed” is the most complex. Use the “Compilation:” menu to switch between the different semantics and re-compile the program in each case. Observe how the complexity of the reaction graph (in the “Graph” sub-tab) changes. You can also use the “Text” sub-tab to compare the numbers of different species and reactions produced in each case. Notice that in the simpler models, multiple reactions are condensed into a single step. How many reactions are there in the “Detailed” model? (Count a reversible reaction as two reactions.)
6. The DSD tool also includes a deterministic simulator which constructs and solves an Ordinary Differential Equation representation of the system. Select “Deterministic” from the “Simulation:” drop-down menu, then re-run the simulation. Run the deterministic and stochastic simulations for the various compilation options, taking note of the difference in performance.

II: State space analysis and modular programming

1. Load the “Two-domain transducer” example program and select the “Infinite” semantics from the “Compilation:” menu. This simple example implements a DNA gate which converts a “<t^ x>” input strand into a “<t^ y>” output strand. Compile the reactions and observe the inputs and outputs on the reaction graph in the “Compilation” tab.
2. Visual DSD supports construction and visualisation of continuous-time Markov chains which represent the “state space” of a system. This can be thought of as a state transition system where the states represent a particular set of species populations and the transitions correspond to reactions involving the DNA species. Switch to the “Analysis” tab on the right-hand side and select the “Graph” sub-tab. Use the “Analyse” button to compute the state space and investigate the resulting graphical visualisation. The “Graph” sub-tab includes numerous options to adjust the layout of the graph. The “Visualise” sub-tab displays just the initial state and any states from which no further reactions are possible (“terminal” states). This is helpful if the graph is too large to see clearly.
3. In the graph, the initial state of the system is highlighted with a thick black outline, and any terminal states are highlighted with a thick red outline. How many terminal states are there in this case? Check that the “<t^ y>” output strand is produced in the terminal state(s) of this system.
4. The DSD language supports modular programming to allow reuse of common design patterns. In this example, the definition of the “T(N,x,y)” module produces N copies of a transducer gate which turns “<t^ x>” into “<t^ y>”. This is instantiated as “T(1,x,y)” in the last line of the code. Add a second transducer of the form “T(1,y,z)” to the system, by placing it within the parentheses on the final line, separated from the other transducer declaration by a vertical bar (for parallel composition). The last line should now read (<t^ x> | T(1,x,y) | T(1,y,z)) The resulting system should turn “<t^ x>” into “<t^ y>” using the first transducer and then turn “<t^ y>” into “<t^ z>” using the second transducer. To check this, recompute the state space graph. Check that the “t z” output appears in the terminal state.

5. The “new a” declaration within the definition of the “T(N,x,y)” module ensures that each instantiation of the transducer module chooses a fresh domain “a”. This prevents crosstalk between gates and allows them to function correctly. Delete the “new a” declaration from your modified program and re-compute the state space. How many terminal states (shown in red) are there now? Use the “Visualise” sub-tab to investigate them and try to identify which is the desired terminal state and which is the undesired one, bearing in mind the functionality of the transducer gate (Hint: in the Terminal state, all of the colour (toehold) domains in the gate complexes should be closed off, i.e. double-stranded).

III: Modelling interference

1. Load the “Buffered Transducer” example program and set the compilation rules to “Default” and the simulation to “Stochastic”. This program implements another kind of transducer which receives “x” strands and outputs “y” strands.
2. Compile the program, view the reaction graph and note down the number of species and reactions (from the “Text” sub-tab of the “Compilation” tab). Run a stochastic simulation to familiarise yourself with the correct behaviour.
3. Visual DSD allows the modelling of interference via “leak reactions”. These formalise a particular kind of unwanted interaction between strands and gates. Enable leaks by checking the “Leaks” checkbox in the “Options:” drop-down menu and recompile the program. How do the compilation time and the size of the reaction graph compare to the non-leak case? Run a simulation and compare the behaviour to the non-leak case. In this case, enabling leaks has qualitatively changed the behaviour of this system by allowing spurious additional “y” strands to be emitted.
4. Visual DSD supports a just-in-time (JIT) compilation and simulation algorithm which allows larger systems to be simulated without pre-computing all of the possible reactions, which can take a long time (as in the case of this leak example). Without disabling leaks, select “JIT” from the “Simulation:” drop-down menu to enable the JIT simulator, then click “Compile” to recompile the system. Note that the “Compilation” tab is not populated in this case, because the JIT compiler does not pre-compute the full reaction network. Run the simulator and check that the time series is similar to that produced by the stochastic simulator for this example. When the simulation finishes, the “Compilation” tab will be populated as normal.
5. The JIT compiler only calculates new species and reactions as and when they are needed. Compare the number of species and reactions in the reaction network produced by the “JIT” compiler to the numbers from the full reaction network computed when “Stochastic” simulation was selected. The discrepancy is due to the fact that leak reactions have very low rates and hence many of the reactions in the full reaction network never actually occur. The JIT simulator allows us to efficiently handle very large reaction networks, such as those produced using leaks.

Answers

Part I, number 4: “directive duration 2000.0 points 500”.

Part I, number 5: There are 19 reactions, if you count a reversible reaction as 2 reactions.

Part I, number 6: The Detailed model runs really slowly under stochastic (because the explicit migration reactions are very fast and bog down the simulation) but fairly well in deterministic mode. The other models are simple enough to run very fast in both modes.

Part II, number 3: There is 1 terminal state.

Part II, number 5. There are 2 terminal states. Removing “new a” allows strands from the first transducer gate to interfere with the second transducer gate, allowing it to produce its output prematurely. Thus in the incorrect terminal state there are gates still waiting to receive input whereas in the correct terminal state all the gate structures have been completely closed off (they are totally double stranded).

Part III, number 3: The graph will be much bigger with leaks enabled, and much slower to compile.

Part III, number 5: The numbers for the JIT case should be lower. The numbers are different for each simulation run because the JIT simulation is stochastic, so the species and reactions encountered may be different each time.