Chapter 3

A MATLAB® Simbiology® Toolbox for Circuit Behavior Prediction in TX-TL and Concurrent Bayesian Parameter Inference

3.1 Introduction and Background

The use of computer-aided design (CAD) tools, such as SPICE (Simulation Program with Integrated Circuit Emphasis, [46]) for electric circuit design, decreases the design iteration time in engineering disciplines. We have developed an analogue of such tools for the TX-TL prototyping platform, in the form of a MATLAB® toolbox called `txtlsim` that allows for easy specification, characterization and simulation of genetic circuits.

The use of CAD tools in systems and synthetic biology is not a novel idea. Some examples of simulation software include the TABASCO simulator [40], COPASI [31], ProMot [44], Cello [48] and bioscrape [62]. TABASCO allows for fast stochastic simulation of gene regulatory circuits at the single molecule and single base pair resolution while not trading off too much speed. It does this by employing a dual architecture that allows for switching between modeling base pair resolution reactions and species level reactions. COPASI is a general purpose simulator that allows for the simulation of both stochastic and deterministic models, and even for hybrid models where low copy number species are simulated stochastically, and all other species deterministically. The Process Modeling Tool (ProMoT) employs a unique method for formulating genetic circuits in a *composable* format, with well defined biochemical signal carriers between the parts [16,44]. Signal carriers take the form of polymerase per second (PoPS), ribosomes per second (RiPS), transcription factor per second (FaPS) and inducer or cofactor signal per second (SiPS). Each part has a defined set of inputs and output terminals, and can only be composed with another part with a corresponding set of terminals. Cello is an example of an electronic design automation tool that takes a desired function as an input, and draws upon a library of Boolean logic gates to generate candidate circuits that perform that function. Finally, bioscrape is a tool developed for performing fast stochastic simulations with time delays, cell lineage tracking and Bayesian parameter inference for general genetic circuit models.

Due to the often complementary nature of the tools available for simulation, inference and analysis, it is desirable to have a way to transfer models between these tools. The Systems Biology Markup Language (SBML) is a widely adopted XML (eXtensible Markup Language) based format for representing biochemical networks. The use of such an information standard for specifying biochemical networks has other advantages: it reduces the chance of old models being lost when the simulator they were written for are no longer supported, and makes it easier for users to parse and understand models written by other researchers, possibly using other tools [34]. The SBML specification is divided into 'levels', where level one specifies a hierarchy of objects that can be used to specify a biochemical network: a model, the comprising compartments, reactions, species (reactants and products in the reactions), parameters and rules. The subsequent levels are intended to implement other functionalities associated with the base network, such as support for MathML and metadata.

In this chapter we describe `txtlsim` in enough detail for the reader to get a sense of the main capabilities of the toolbox. `txtlsim` is written using MATLAB® Simbiology®, which in turn is modeled after SBML. Indeed Simbiology® defines models, compartments, reactions, species, parameters, rules and events as classes, and provides a rich set of methods and properties associated with them. In `txtlsim`, DNA and individual species to be added to TX-TL can be specified using a set of symbolic specification rules, and the toolbox generates a deterministic mass action model of the gene regulatory network expected to exist in TX-TL under the specified conditions. A typical TX-TL model, specified at the resolution of whole DNA, mRNA and protein species is composed of transcription, translation, RNA degradation, regulatory mechanisms and the inactivation of the ability of TX-TL to express genes. Optionally, linear DNA and protein degradation can be included. Furthermore, other special mechanisms, like sigma factor action or RNA-mediated transcriptional attenuators, may also be included [65].

We highlight several features of `txtlsim`. Firstly, this toolbox requires only a few lines of code to generate a complex chemical reaction network that models the reactions in TX-TL. In lower level specifications, such as Simbiology®, bioscrape [62] or even simply raw ODEs, this would amount to several tens to over a hundred equations that would need to be manually specified and processed. The key reason for the need for this complexity is that the toolbox is able to model the consumption of limited nucleotides and amino acid species, and the loading of the finite catalytic machinery (RNA polymerases, ribosomes, RNases, transcription factors etc). The consumption and degradation of nucleotides and amino acids is thought to underlie the inactivation of the gene expression capability, and is therefore important to model for capturing the full curves of TX-TL reactions. Coupling between different parts of a circuit, via the loading of enzymatic resources [29] or regulatory elements has been shown to introduce unintended interactions between parts of genetic circuits in both TX-TL and *in vivo* [13]. These types of retroactivity or loading effects are automatically and simply incorporated into `txtlsim`, at least with respect to the species that exist in the toolbox, by virtue of the fact that we use mass action models. In fact, a more general property holds: the models built using the toolbox are extensible in the sense that once a species exists, if a new type of interaction is added that relates to that species, none of its previous interactions need to be modified explicitly. Another feature of `txtlsim` worth noting is that the models generated with it can be converted into SBML, and may be exported into any other SBML compatible environment for analysis. The final feature is the MCMC based Bayesian parameter inference capabilities incorporated into `txtlsim` in the form of a sub-toolbox called mcmc_simbio. The main feature distinguishing this from existing parameter inference tools is the ability to perform concurrent

Bayesian parameter inference on a *set* of model-experimental data pairs that contain parameters with common identities. This feature is built around a MATLAB® implementation [28] of the affine invariant ensemble MCMC sampler [25] for generating the parameter posterior distributions given the model, data and experimental setup. Our wrapper adopts this sampler to estimate parameter distributions for models written generically in Simbiology®, which in turn is able to import SBML models, and can therefore be used for parameter inference with a large class of models. We note that the study in [35] looked at the issue of concurrent parameter inference (referred to as *consensus* inference there), but only used optimization procedures to estimate point estimates of parameters. Thus, their method does not provide the main advantage of Bayesian inference: insight into parameter identifiability. Indeed, with the concurrence feature, it becomes possible to inform parameters from multiple model-data pairs, potentially improving identifiability. This, in turn, increases the value of using Bayesian inference to study the identifiability properties of model parameters.

In this chapter, we describe the modeling framework, usage and architecture of the toolbox, along with the parameter inference capabilities included in the mcmc_simbio subtoolbox. In Section 3.2, we describe the user end code for setting up a TX-TL model for tetR mediated negative autoregulation. In Section 3.2.1, we elaborate on the choice of the chemical reactions implemented in the toolbox. In Section 3.3 we characterize the parts of an incoherent feedforward loop motif and compare model predictions to experimental data. Next, in Section 3.4, we discuss the software architecture that enables the automatic generation of the biochemical network. Finally, we discuss multi-experiment concurrent parameter inference capabilities of mcmc_simbio in Section 3.5.

3.2 An Overview of the `txtlsim` Toolbox

In this section, we describe the `txtlsim` toolbox in some detail. The code snippet shown below depicts what a user would write to set up the negative autoregulation circuit, in which a repressor represses its own expression, along with that of a reporter.

```
% set up extract and buffer tubes (Simbiology `Model' objects) with parameters
    from a configuration file identified to a particular extract batch.
tube1 = txtl_extract('E1');
tube2 = txtl_buffer('E1');
tube3 = txtl_newtube('negative_autoregulation');
% add DNA specifying a negative autoregulation circuit
txtl_add_dna(tube3, 'ptet(50)', 'UTR1(20)','deGFP(1200)', 1, 'plasmid');
txtl_add_dna(tube3, 'ptet(50)', 'UTR1(20)','tetR(1200)', 0.2, 'plasmid');
% combine tubes, add inducer, 'run' the experiment and visualize results
Mobj = txtl_combine([tube1, tube2, tube3]); % Simbiology Model object
txtl_addspecies(Mobj, 'aTc', 500); % add inducer
simData = txtl_runsim(Mobj, 12*60*60); % Simulate 12 hours of trajectories
txtl_plot(simData, Mobj);
```

The set of commands above closely mimic the actual experimental protocol of setting up the reaction. The functions `txtl_extract` and `txtl_buffer` access extract and buffer parameter configuration files, specified by the input string `'E1'` here, to set up two Simbiology model objects called `tube1` and `tube2` respectively, which are model objects containing extract and buffer specific parameters and species. The configuration files are user defined, and the parameters they contain can come from the literature, or from parameter inference performed on experimental data.

Next, the `txtl_newtube` and `txtl_add_dna` commands are used to initialize a new model object and add different DNAs to the tube respectively. In its most common use case, the `txtl_add_dna` command allows for specification of promoter, untranslated region and coding sequence to form a transcriptional unit on the specified DNA, along with the concentration of the DNA added, and whether it is a linear fragment or plasmid DNA. For example, in the first call to `txtl_add_dna`, the promoter, ribosome binding site (RBS) and coding sequence (CDS) are specified by the strings `'pOR2OR1'`, `'UTR1'` and `'tetR'` respectively. These strings, each describing a component of the transcriptional unit, are used to access a library containing code and parameter files associated with the respective components. These component files specify all the reactions and species associated with the component, and allow for the modular composition of these components into circuits.

The txtl_combine command is used to combine the three tubes into a single model object, Mobj, which is subsequently simulated using the txtl_runsim command.

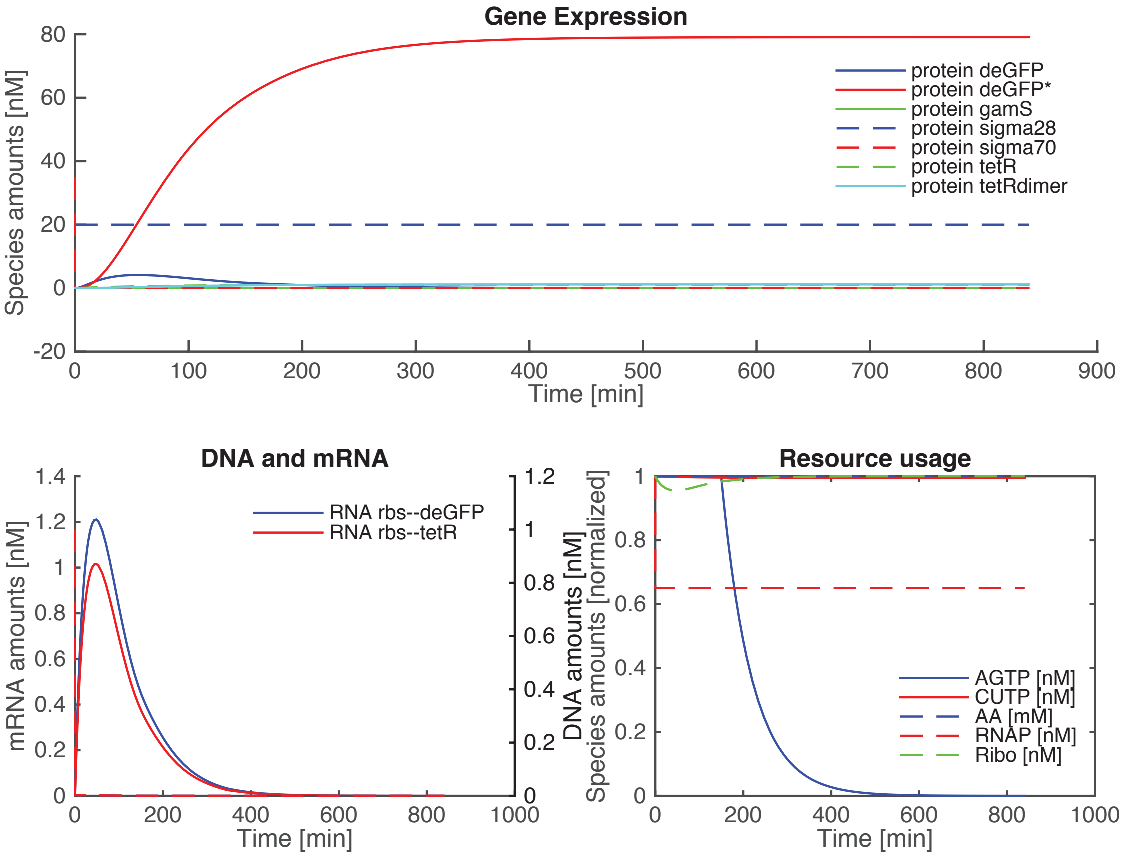

Figure 3.1: Standard output of the TXTL toolbox.

Figure 3.1 shows the result of the txtl_plot command, which is arranged into three panels. The top panel shows the protein species that exist within the system. The protein deGFP* is the folded GFP. Bottom left plot shows RNA (solid) and DNA (dashed) dynamics. RNA rises before repression by TetR causes transcription to stop. The bottom right plot (normalized to 1) shows that the AGTP species degrades after about 3 hours ([49] Figure 1B). The other species we can observe are CUTP, ribosomes, amino acids and RNA polymerases.

3.2.1 The Modeling Framework of the `txtlsim` Toolbox

Here we describe the typical reaction network generated by `txtlsim` when a transcriptional unit (TU) is expressed. More complex networks made out of multiple TUs interacting via transcription factor (TF) mediated regulation are simply iterations of this canonical network, but coupled via catalytic and consumable resources, and the relevant regulatory interactions.

We begin with a description of the species naming convention used in the toolbox. The species in the toolbox may be divided into five broad categories: DNA, mRNA, proteins, miscellaneous species like inducers or nucleotides, and the biochemical complexes formed by combining these in defined ways. To avoid a combinatorial explosion, not every possible species that can exist is created, and instead, the toolbox uses the user inputs and corresponding reactions to define the set of species to be created. The species follow a strict naming convention, allowing for the use of regular expressions in parsing the name stings, and for making the decisions required for the creation of the chemical reaction network underlying a given model. Example conventions for DNA, RNA and proteins are given in Table 3.1

Table 3.1: Species naming conventions

Species Type	Convention	Example

DNA	`DNA <promspec>--<utrspec>--<cdsspec>`	`'DNA thio-junk-ptet--utr1--tetR'`
		`'DNA ptet--utr1--tetR-lva'`

RNA	`RNA <utrspec>--<cdsspec>`	`'RNA utr1--tetR'`
		`'RNA att1-utr1--tetR-lva'`

protein	`protein <cdsspec>`	`'protein tetR'`
		`'protein tetR-lva'`

Here, `promspec`, `utrspec` and `cdsspec` are the promoter, untranslated region (UTR) and coding sequence specifications respectively. Some examples of the variations of these specifications are shown in Table 3.1. The specifications are separated by the long hyphen '–', and within each specification, we may have various types of modifiers, such as `junk` DNA on the promoter to protect against DNA degradation, attenuator RNA in the untranslated region [65] or `lva` protein degradation tags on the coding sequence. The miscellaneous species include inducers like anhydrotetracycline (aTc) or Isopropyl beta-D-1-thiogalactopyranoside (IPTG), core species like ribosomes (`ribo`), RNA polymerases (`RNAP`), `RecBCD` and `RNase` nucleases, etc., and resources like amino acids (`AA`) and grouped nucleotide species (`AGTP`, `CUTP`).

We now turn to a discussion of the reactions set up by the toolbox. The three main processes that almost always get set up for every DNA specified by the user are transcription, translation and RNA degradation. DNA degradation via the RecBCD nuclease happens only to linear DNA fragments, and can be reduced by adding the protein GamS to the system, which sequesters RecBCD [21, 61]. Protein degradation is only active when the ClpXP protease is present in the system, and the protein to be degraded is tagged with a degradation tag. Other conditional behaviors include TF mediated promoter occlusion or activation, protein dimerization, maturation, binding to small molecules like inducers, and non-coding RNA based regulation. These behaviors are included in the biochemical reaction network when the relevant DNA or individual molecule species are specified as inputs to the system. Figure 3.2 shows the general set of reactions associated with each DNA that is specified as an input using the `txtl_add_dna` command.

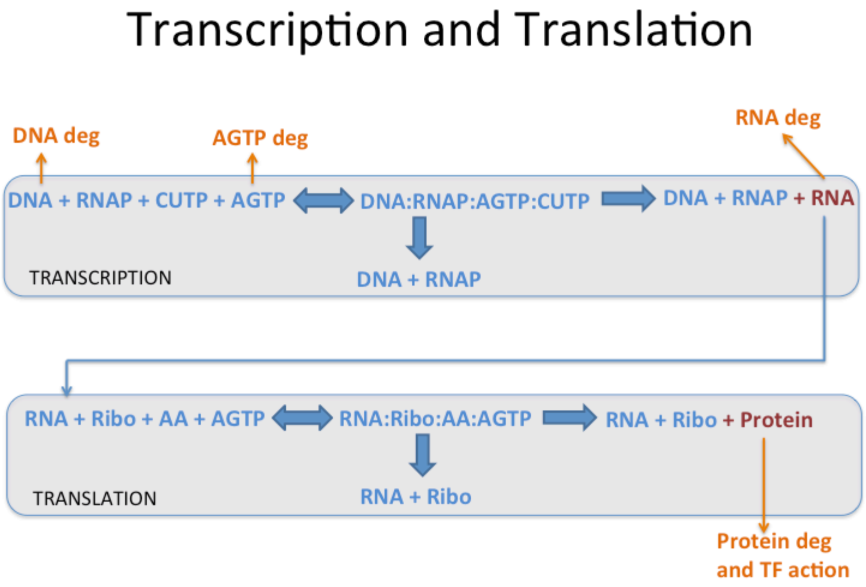

Figure 3.2: A high level description of the mechanics present in the toolbox for each transcriptional unit.

Transcription is modeled using the equations

$$\begin{aligned}
\mathrm{RNAP} + \mathrm{DNA} &\rightleftharpoons \mathrm{RNAP{:}DNA}, && \text{TX machinery binding to DNA,}\\
\mathrm{RNAP{:}DNA} + \mathrm{AGTP} &\rightleftharpoons \mathrm{AGTP{:}RNAP{:}DNA}, && \text{nucleotide binding,}\\
\mathrm{RNAP{:}DNA} + \mathrm{CUTP} &\rightleftharpoons \mathrm{CUTP{:}RNAP{:}DNA}, && \text{nucleotide binding,}\\
\mathrm{AGTP{:}RNAP{:}DNA} + \mathrm{CUTP} &\rightleftharpoons \mathrm{CUTP{:}AGTP{:}RNAP{:}DNA}, && \text{nucleotide binding,}\\
\mathrm{CUTP{:}RNAP{:}DNA} + \mathrm{AGTP} &\rightleftharpoons \mathrm{CUTP{:}AGTP{:}RNAP{:}DNA}, && \text{nucleotide binding,}\\
\mathrm{CUTP{:}AGTP{:}RNAP{:}DNA} &\xrightarrow{k_{tx}} \mathrm{RNAP{:}DNA_{term}} + \mathrm{mRNA}, && \text{mRNA production,}\\
\mathrm{CUTP{:}AGTP{:}RNAP{:}DNA} &\xrightarrow{\left(\frac{L_m}{4}-1\right)k_{tx}} \mathrm{RNAP{:}DNA}, && \text{consumption, } L_m = \text{RNA length,}\\
\mathrm{RNAP{:}DNA_{term}} &\longrightarrow \mathrm{RNAP} + \mathrm{DNA}, && \text{termination.}
\end{aligned} \tag{3.1}$$

The catalytic machinery of transcription is lumped into a single species, denoted RNAP. It is assumed to encompass RNA Polymerases, sigma factors, and other cofactors, but not transcription factors, whose binding will be modeled explicitly. The consumable nucleotide species ATP and GTP are lumped into a single species AGTP, and CTP and UTP are lumped into a species denoted CUTP. After the binding of the catalytic and consumable species, the production of mRNA itself is divided into two reactions, an mRNA production reaction and a nucleotide consumption reaction. As its name suggests, the consumption reaction simply uses up the nucleotide species AGTP and CUTP, without producing mRNA. The rate of this reaction is a multiple of the transcription reaction rate, with a scaling determined by the mRNA length in bases, L_m, so that the stoichiometry of nucleotide consumption and mRNA production is correct. This modeling choice is discussed at length in Chapter 4, and briefly in Appendix 3.A. At the end of mRNA production, a termination complex $\mathrm{RNAP{:}DNA_{term}}$ forms, which then dissociates into RNAP and DNA in a separate reaction.

The reduced equations for translation,

$$\begin{aligned}
\mathrm{Ribo} + \mathrm{mRNA} &\rightleftharpoons \mathrm{Ribo{:}mRNA}, && \text{ribosome binding to mRNA,}\\
\mathrm{Ribo{:}mRNA} + \mathrm{AA} &\rightleftharpoons \mathrm{AA{:}Ribo{:}mRNA}, && \text{resource binding,}\\
\mathrm{AA{:}Ribo{:}mRNA} + 2\ \mathrm{AGTP} &\rightleftharpoons \mathrm{AA{:}AGTP^2{:}Ribo{:}mRNA}, && \text{resource binding,}\\
\mathrm{AA{:}AGTP^2{:}Ribo{:}mRNA} &\xrightarrow{k_{tl}} \mathrm{Ribo{:}mRNA_{term}} + \mathrm{protein}, && \text{protein production,}\\
\mathrm{AA{:}AGTP^2{:}Ribo{:}mRNA} &\xrightarrow{(L_p-1)k_{tl}} \mathrm{Ribo{:}mRNA}, && \text{consumption, } L_p \text{ = protein length,}\\
\mathrm{Ribo{:}mRNA_{term}} &\longrightarrow \mathrm{Ribo} + \mathrm{mRNA}, && \text{termination,}
\end{aligned} \tag{3.2}$$

look similar to those for transcription. We note that on average it takes two ATP and two GTP per amino acid (AA) residue, leading to the binding and consumption reactions shown below.

RNA degradation is mediated by RNases, and is implemented as an enzymatic reaction,

$$\begin{aligned}
\mathrm{RNase} + \mathrm{mRNA} &\rightleftharpoons \mathrm{RNase{:}mRNA}, && \text{RNase binding to mRNA,}\\
\mathrm{RNase{:}mRNA} &\longrightarrow \mathrm{RNase}, && \text{degradation.}
\end{aligned} \tag{3.3}$$

Similar binding and degradation reactions are set up for mRNA in its various bound forms, such as $\mathrm{Ribo{:}mRNA}$, $\mathrm{AA{:}Ribo{:}mRNA}$, $\mathrm{AA{:}AGTP{:}Ribo{:}mRNA}$, $\mathrm{AA{:}AGTP^2{:}Ribo{:}mRNA}$ and $\mathrm{Ribo{:}mRNA_{term}}$, which result in the degradation of the mRNA and return of the remaining complexed species to the species pool.

Apart from these three main mechanisms, we also model RecBCD mediated linear DNA degradation as an enzymatic reaction, the sequestration of RecBCD by the GamS protein, ClpXP mediated degradation of tagged proteins and transcription factor mediated regulation. Other interactions, for example kinase-phosphatase action, RNA attenuator mediated transcriptional regulation and explicit sigma factor function, can also be included if desired. For brevity, we only list the transcriptional repression and induction reactions here. Repression by the dimerizable protein TetR and its sequestration by the inducer anhydrous tetracycline (aTc) is modeled as

$$\begin{aligned} 2\ \mathrm{TetR} &\rightleftharpoons \mathrm{TetR}_{\mathrm{dimer}}, && \text{repressor dimerization,} \\ \mathrm{DNA} + \mathrm{TetR}_{\mathrm{dimer}} &\rightleftharpoons \mathrm{DNA{:}TetR}_{\mathrm{dimer}}, && \text{DNA sequestration,} \\ 2\ \mathrm{aTc} + \mathrm{TetR}_{\mathrm{dimer}} &\rightleftharpoons \mathrm{aTc}_2\mathrm{:TetR}_{\mathrm{dimer}}, && \text{DNA sequestration.} \end{aligned} \tag{3.4}$$

In Section 3.4, we discuss the software architecture that allows for the automatic generation of these reactions and the interactions between them without the need for the user to specify them explicitly.

3.3 Part Characterization and Circuit Behavior Prediction

In this section, we discuss an example involving the characterization of the parts of a type one incoherent feedforward loop (IFFL), followed by the prediction of the behavior of the IFFL in TX-TL using `txtlsim`, and comparison to experimental data. We begin by parameterizing the model's core mechanics using parameters drawn from the literature. We then decompose the behavior of the IFFL into five distinct parts, and estimate part parameters by fitting models of each part to corresponding experimental data. Finally, we use the characterized parts to predict the behavior of the IFFL under a variety of experimental conditions, and compare the computational predictions with experimental data.

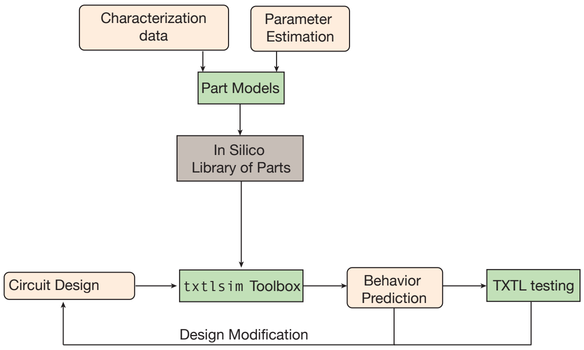

Figure 3.3: The general workflow of using CAD software like `txtlsim` for circuit prototyping. After a library of characterized parts is built, circuit designs can be tested *in silico* and *in vitro*, and modified to fit the design needs. This process can also help refine the models by comparing the model behavior to *in vitro* behavior.

3.3.1 Core Parameters

The parameters in the system come from the literature, and from parameter estimation carried out using experimental data collected in our lab. For parameters from the literature, the main sources are [37] and [60] . Reference [37] gives us the transcription elongation rate of about 1 nt s^{-1}, and a 4 AA s^{-1} lower bound on the translation elongation rate. It finds an mRNA degradation half life of 12–14 min (which we reproduce in Figure 3.5 (i)), and notes that the degradation machinery does not get saturated even when there is 200 nM of mRNA in the system. Furthermore, the following features are observed, which we reproduce in the toolbox for characterization purposes: 30 nM of plasmid DNA gives an approximate steady state of 30 nM of mRNA, 1 µM of protein is accumulated in 1 h, and the accumulation rate decays exponentially over the next 9 hours, with an eventual maximum expression level of about 10 µM (Figure 3.4).

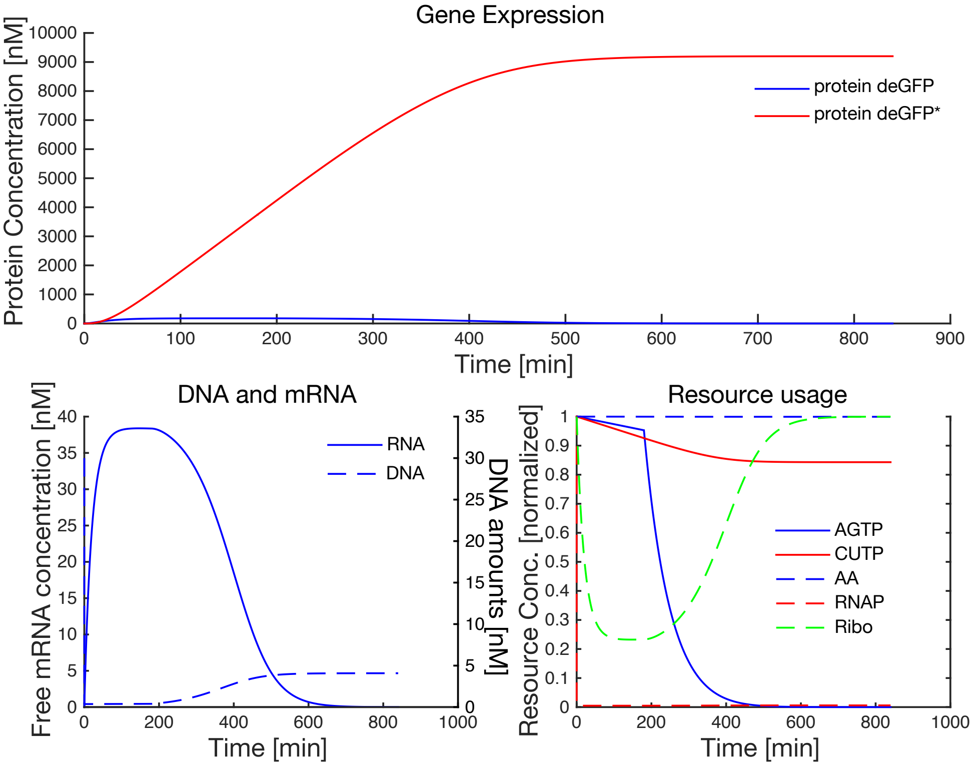

Figure 3.4: Constitutive GFP expression after core parameters were set to values from the literature. When 30 nM of constitutively expressing deGFP reporter plasmid DNA is expressed in TX-TL, about 10 µM of deGFP produced in 10 h, about 30 nM of mRNA steady state is reached, and AGTP starts degrading at about three hours.

The concentration of RNAP and Ribosomes and the Michaelis-Menten constant for transcription is given in Table 1 of Karzbrun et al. [37] as 30 nM, >30 nM and 1-10 nM respectively. Reference [49] shows that ATP levels start to fall exponentially at about three hours, giving us the degradation dynamics of ATP in the system. This is implemented in the toolbox using a Simbiology® event. Reference [60] provides the concentrations of ATP and GTP at 1.5 mM in TX-TL, those of UTP and CTP at 0.9 mM and an AA concentration of 1.5 mM. Figure 3.5 (ii) shows a comparison of experimental results form [37] and the simulation results from the toolbox, for the constitutive expression of GFP when plasmid DNA is varied from 5 nM to 30 nM. Some parameters, like the forward and reverse rates of the binding of amino acids and nucleotides, are difficult to design characterization experiments for, and so we simply fixed them to values that allowed the model to give good agreement with the literature and the experimental data that we collected. These parameters are generally non-identifiable, and the behavior of the model tends to be insensitive to variations in their values over a broad range of values.

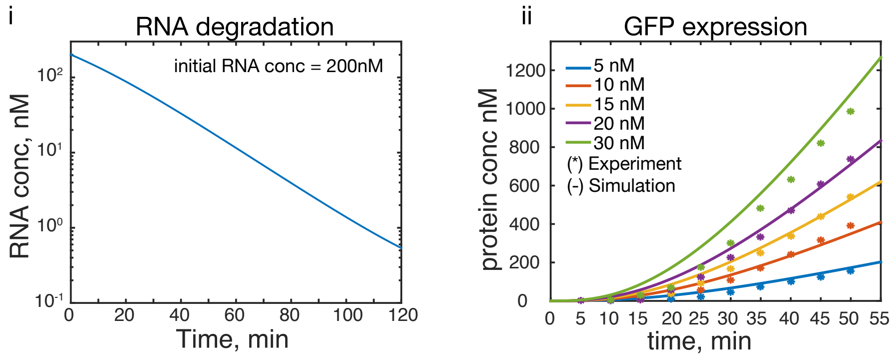

Figure 3.5: (i) RNA degradation half life of about 17 minutes agrees with the numbers in [37] (12 min) and [12] (20 min). (ii) Constitutive GFP expression after core parameters were set to values from the literature. The simulation results compared to the data from [37].

In the next section, we describe how we estimated parameters for the parts of an IFFL, before using the part models to predict the behavior of the IFFL in various experimental conditions.

3.3.2 IFFL Part Specific Parameters

The IFFL is a circuit in which an activator transcription factor simultaneously activates a reporter protein and a repressor transcription factor. The reporter protein is also repressed by the repressor. Owing to the fact that activation only requires the production of one protein (the activator), and repression requires the production of two proteins (the activator, followed by the repressor), repression of the reporter is delayed with respect to activation. In cells, where there is dilution present, this mechanism leads to the reporter concentrations showing a pulse. In TX-TL, without active protein degradation, one simply observes a cessation of reporter protein accumulation that occurs sooner than that which would be expected due to the inactivation of TX-TL. Figure 3.6 (Bi) shows a schematic of the IFFL, where the circles represent proteins, pointed arrows show activation (lasR to tetR and lasR to deGFP) and blunt arrows show repression (tetR to deGFP). The inducers 3OC12 (a type of N-acyl homoserine lactone, abbreviated AHL) and anhydrous tetracycline (aTc) activate lasR and sequester tetR respectively, and are shown with green and red arrows (respectively). The lasR protein is under the control of the constitutively expressing pLac promoter, the tetR protein is under the control of the pLas promoter, which is activated by LasR in the presence of 3OC12. The deGFP reporter protein is under the control of a combinatorial promoter, which is only active when activated lasR is present and tetR is absent (or sequestered by aTc). The characterization of the parts of the IFFL was performed using five experiments: the constitutive expression behavior of the pTet and pLac promoters, tetR mediated repression of the pTet promoter, aTc induction, and finally induction via activated lasR. These experiments are summarized in Table 3.2 and the results of the experiments, along with model fitting, are shown in Figure 3.6 (A). All the experiments in Figure 3.6 were performed using plasmid DNA.

Each of the five characterization experiments have subsets of parameters in the model that are naturally associated with them. For instance, the constitutive expression of pTet Figure 3.6 (Ai) informs the dissociation constant for the pTet DNA and RNAP. The Ribosome to RNA binding dissociation constant and the amino acid binding constant were not taken from any literature source, so we chose to estimate these using the first estimation data

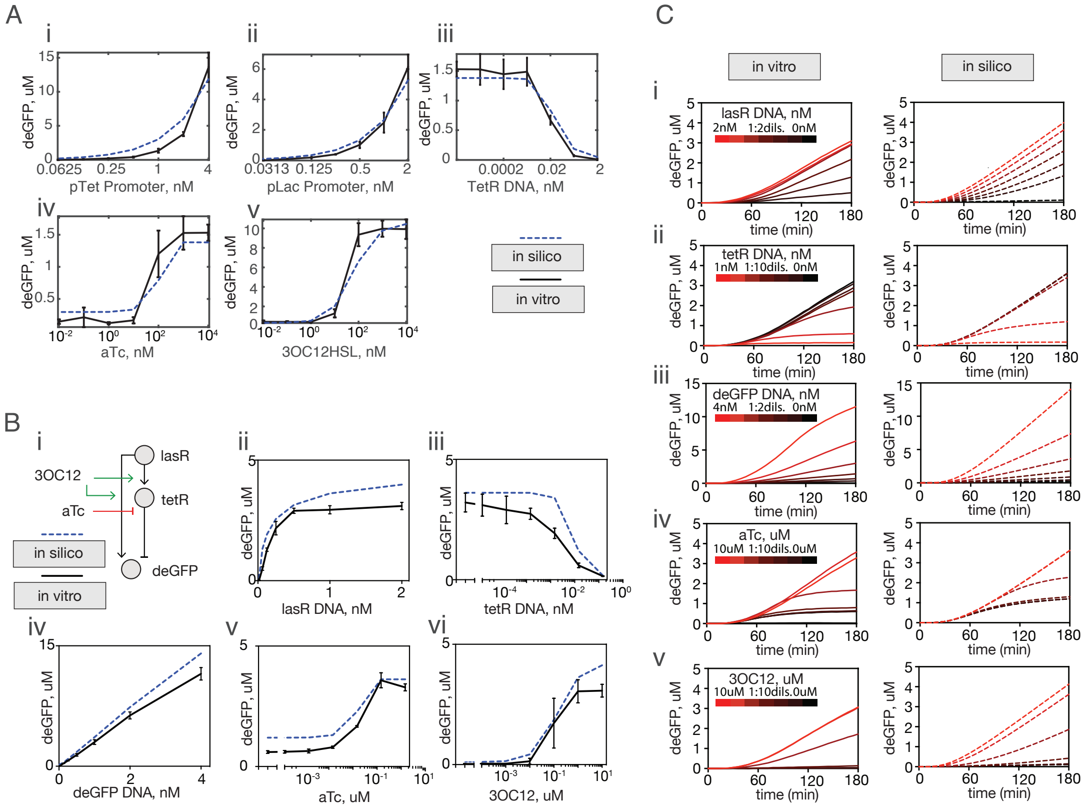

Figure 3.6: Incoherent feedforward loop (IFFL) characterization and behavior prediction. (A) Experiments were performed on parts of the IFFL, and the reporter expression time course data were fit to corresponding models to estimate parameters. Panels (i - v) show the endpoints of the experimental data and the fitted trajectories plotted on the same axes for comparison. Vertical lines show error bars from replicate data. The experimental conditions are described in Table 3.2. (B, C) Comparing predictions from the characterized model to experimental data. (B) shows the endpoints of the model trajectories and the experimental data plotted on the same axes for the five experimental variations tested, and (C) shows the same experiments and model predictions, but for the full time course trajectories. Details of the experimental conditions are in the main text.

Table 3.2: Description of panels in Figure 3.6 (A)

Panel	Experiment	Parameter(s) Estimated	Associated Reactions and Notes
i	Constitutive expression deGFP under a TetR responsive promoter: pTet-UTR1-deGFP at 4, 2, 1, 0.5, 0.25, 0.125 and 0.0625 nM to sequester any native LacI.	$K_{d,pTet}$, $K_{d,ribo}$, $K_{d,AA}$	RNA polymerase binding to the promoter (DNA), Ribo binding to ribosome binding site (RNA) and Amino acids and AGTP binding to the ribosome - RNA complex. The first parameter can be used as a measure of promoter strength, and the other two parameters were estimated here, and fixed to the estimated values during the estimations in panels ii - v.
ii	Constitutive expression of deGFP under a LacI responsive promoter: pLacO1-UTR1-deGFP at 2, 1, 0.5, 0.25, 0.125, 0.0625 and 0.0313 nM. IPTG at 1 mM to sequester any native LacI.	$K_{d,pLac}$	RNA polymerase binding to the promoter (DNA). This parameter can be used as a measure of promoter strength.
iii	pTet repression; jointly with (iv). pTet-UTR1-deGFP at 1 nM. pLac-UTR1-deGFP varied at 2, 0.2, 0.02, 0.002, 0.0002, 0.000 02 and 0.000 002 nM. IPTG at 1 mM to sequester any native LacI.	$K_{d,tdim}$, $K_{d,trep}$, $K_{d,aTc}$	tetR dimerization, ptet DNA sequestration and aTc binding to tetR dimer. Estimation performed jointly with the data in panel iv.
iv	tetR induction (jointly with iii). pTet-UTR1-deGFP at 1 nM. pLac-UTR1-deGFP at 0.1 nM. aTc varied at 10, 1, 0.1, 0.01, 0.001, 0.0001 and 0.000 01 µM. IPTG at 1 mM to sequester any native LacI.	$K_{d,tdim}$, $K_{d,trep}$, $K_{d,aTc}$	tetR dimerization, ptet DNA sequestration and aTc binding to tetR dimer. Estimation performed jointly with the data in panel iii.
v	3OC12 induction of pLas: pLac-UTR1-LasR at 1 nM, pLas-UTR1-deGFP at 1 nM, 3OC12 varied at 10, 1, 0.1, 0.01, 0.001, 0.0001 and 0.000 01 µM. IPTG at 1 mM to sequester any native LacI.	$K_{d,OC12}$, $K_{d,LasLeak}$, $K_{d,LasAct}$, $K_{d,pLas}$	3OC12 binding to lasR, RNAP binding to plas DNA, [OC12:lasR] binding to plas DNA and RNAP binding to activated plas DNA

from this set, and fix these for all subsequent fitting and simulation. Estimation was carried out using the Simbiology® toolbox for MATLAB®, where we used the Levenberg-Marquardt algorithm to solve a non-linear least squares fitting problem to perform the parameter fitting.

3.3.3 Model Predictions

Using these estimated parameters, we built an IFFL *in-silico* and compared its behavior to its *in vitro* analogue. The results are shown in Figure 3.6 (B, C), which show the endpoint expression and the full time course trajectories as a function of experimental conditions respectively. The nominal experimental conditions for the IFFL in Figure 3.6 (Bi) were as follows. IPTG was added at $1\,\mathrm{mM}$, making the pLac promoter constitutive by sequestering native LacI in the extract. The LasR inducing AHL, 3OC12, was at $1\,\mu\mathrm{M}$. The constitutive activating plasmid pLac-UTR1-lasR was at $1\,\mathrm{nM}$. The repressor DNA pLas-UTR1-tetR was at $0.1\,\mathrm{nM}$ and the reporter DNA plastetO-UTR1-deGFP. The tetR inducer aTc was at $10\,\mu\mathrm{M}$, over which it is toxic to TX-TL. This is the reason why the tetR DNA concentration was kept at a low value of $0.1\,\mathrm{nM}$. At this concentration, there is enough tetR produced to repress pTet almost completely (Figure 3.6 (Aiii)) in the absence of aTc, while still keeping the tetR levels low enough for $10\,\mu\mathrm{M}$ of aTc to fully sequester it. With these nominal conditions, the perturbations shown in the panels (Bii-vi) and (Ci-v) in Figure 3.6 were applied, with the results as shown. We note that the model of the IFFL generated by `txtlsim` and characterized as described in Section 3.3.2 was able to predict the the *in vitro* behavior of the IFFL well.

3.4 Automated Reaction Network Generation

In Section 3.2, we gave an overview of how a user may set up a simple circuit using a few lines of code in `txtlsim`. In this section we go into further details of how the software sets up the model. We start with a walk-through of what each command does, along with a discussion of some of the architectural features of the toolbox, and conclude with a discussion of how the nucleotide and amino acid consumption is modeled for the single step mRNA or protein production models used in the toolbox.

3.4.1 Software Architecture Walk-Through

In this section we cover how the toolbox sets up a model object, and in doing so, highlight various features of the toolbox. Figure 3.7 shows the basic flow of the user level code. We will discuss the function of each of of the commands in the user level code, and in doing so provide an overview of the structure of the toolbox.

The main directory of the toolbox is called `trunk`. The key subdirectories in this di-

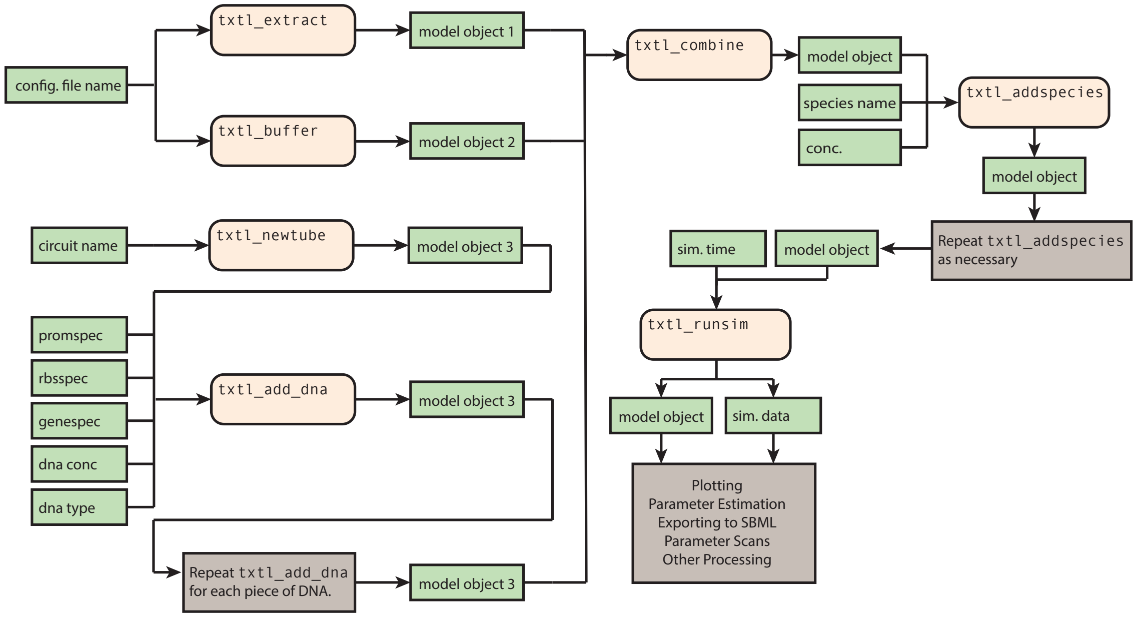

Figure 3.7: Flowchart of the user level code. The **`txtl_add_dna`** command is the main command that is used to specify the DNA to be added to the model. This allows for all the reactions and species associated with that DNA to be set up in the model. The model is contained in a Simbiology® model class object, and is simulated using the **`txtl_runsim`** command. See main text for mode details.

rectory are shown in Table 3.3. In particular, we draw attention to the `core`, **`config`** and **`components`** directories, which we will be referring to in the code walk-through. The `core` directory contains most of the source code of the network generation part of the toolbox. In contains user end functions like **`txtl_add_dna`** and hidden functions such as **`txtl_mrna_degradation`** or **`txtl_transcription`**. The **`config`** directory contains (.csv) configuration files containing parameters associated with extracts and buffers. In principle, each extract and buffer has its own configuration file, which is populated using characterization data collected in that extract and buffer. The **`components`** directory acts as a library of 'parts'. It contains code (.m) and parameter configuration (.csv) files for genetic circuit parts, like promoters, UTRs and CDSs. Promoters in this library can be of an activatable, repressible or combinatorial (e.g.: pLastetO in Figure 3.6) nature. Some promoters, like the arabinose induced pBAD promoter may be repressed by a transcription factor (AraC in this case) when it is not bound to its inducer, and activated by the transcription factor when it is bound to the inducer. There is currently only one type of UTR, the ribosome binding site component, specified as component files in the toolbox. Antisenseattenuator RNA mediated regulation of transcription, which is also specified via the UTR, is implemented as part of the main source code, and not separate component files. Future releases of the toolbox may separate out this capability into separate component files. Finally, CDSs form the most variable group of component files, and can include reporters, repressors, activators, sigma factors, kinases, phosphatases or proteases, to name a few. All three classes of components can be extended in a straightforward manner to include new components, either as copies of existing files with trivial name changes, of with a small amount of additional work to include capabilities not present in the toolbox.

Table 3.3: Directory structure of the Toolbox

Directory	**Description**
core	Core functions of the toolbox, such as `txtl_add_dna` or `txtl_transcription`
`config`	Extract and buffer configuration files (.csv). These contain parameters like transcriptional elongation rate, or the initial concentration of RNA polymerases or nucleotides corresponding to a given extract.
`components`	Component (promoter, UTR and CDS) files. This directory contains both code (.m) and parameter configuration (.csv) files.
`mcmc_simbio`	MCMC toolbox for Simbiology®. This toolbox allows for Bayesian parameter inference to be performed on the parameters of Simbiology® models concurrently over many model-data set pairings. More details can be found in chapter xx.
`examples`	Examples for the modeling toolbox. Includes examples from constitutive gene expression, to the incoherent feedforward loop and the genetic toggle.
`doc`	Contains the User Manual and associated files.

As mentioned in the overview in Section 3.2, the commands `txtl_extract` and `txtl_buffer` are used to initialize the extract specific parameters and species. These functions set up a `txtl_reaction_config` class object that contains methods and properties to manage most of the core (i.e., non part-specific) parameters in the model. The properties of the `txtl_reaction_config` class object are set by a configuration file stored in the `config` directory.

The command `txtl_add_dna` is the workhorse of the network generation phase of the toolbox, and is discussed in some detail here. It takes a model object as its first input, followed by a promoter specification string `promsepc`, a UTR specification string `utrspec`, a CDS specification `cdsspec`, a numerical DNA concentration input, and a DNA type specification string as inputs (Table 3.4). Generically, a call to this function takes the form,

```
txtl_add_dna(model_object, promspec, utrspec,cdsspec, DNAconc, DNAtype),
```

where the `promsepc`, `rbssepc` and `cdssepc` strings are used to access component files of the same names in the component directory. These files contain all the relevant information pertaining to the promoter, RBS or CDS being specified, including the reactions it is involved in and the associated parameters.

Table 3.4: Inputs to the `txtl_add_dna` command. The parenthetical arguments within the specifications are optional, and if they are not specified, then default values from the component configuration files are used. The DNA concentration can be any nonnegative numerical value, and the DNA type must be either `'linear'` or `'plasmid'`.

Input	**Syntax**	**Example**
`model_object`	Simbiology® model object	`tube3`
`promspec`	string(optional numeric)	`'pOR2OR1(50)'`
`utrspec`	string(optional numeric)	`'UTR1(40)'`
`cdsspec`	string(optional numeric)	`'tetR(650)'`
`DNAconc`	numeric	`20`
`DNAtype`	string	`'linear'`

The `txtl_add_dna` command is called twice: once when the user first specifies the DNA, and a second time when the command `txtl_runsim` is called. In the first call, which happens in a ‘species setup’ mode, most of the species associated with that DNA are specified, and in the second call (‘reactions setup’ mode), the previously specified set of species is used to set up the reactions within the model. The reason for splitting the set up of the species and the reactions is that the specification of many reactions in the toolbox requires knowledge of exactly which version of the reactants are present. Thus it must be ensured that any command that sets up reactions in the system has access to the exact versions of any species that might appear as reactants in the reactions to be set up. If the `txtl_add_dna` command were to attempt to set up reactions during its first call, it would not have access to the promoter, UTR and CDS specifications of subsequent lines of `txtl_add_dna`, and therefore to the versions of the species created due to those specifications. One example where this issue arises is as follows. Consider once again the code snippet for setting up the negative autoregulation circuit shown in Section 3.2. The first call to `txtl_add_dna` involves the `ptet` promoter. One of the reactions in this promoter's component file, `txtl_prom_ptet.m` is the binding of the DNA this promoter is a part of (`DNA ptet--UTR1--deGFP`) to the dimerized tetR protein species. The dimerized tetR species, if present at all, can appear in one of two forms: a form that is not tagged with a protein degradation tag, `protein tetRdimer`, and one that is, `protein tetR-lvadimer`. The version of this species that exists depends on what the string specified by `cdsspec` is: `tetR` or `tetR-lva`. Since the `txtl_add_dna` command specifying this is in a subsequent line, this information is not available to the pTet component file at the time it attempts to set up the reaction in this scenario.

One possibility for the first call to `txtl_add_dna` is that it sets up all the possible versions of the repression reaction, and only the reactions with all reactants with non-zero concentrations have flux through them. While this approach would give the correct system dynamics in principle, it is not scalable as the number of species and reactions would get large quickly, with most of these being unnecessary. A better approach is to only set up the reactions that are actually expected to occur in the system. This approach can be implemented with the two-pass method described above. Specifically, the first set of calls to `txtl_add_dna`, which are the calls explicitly visible in the code snippet, set up all the species that are possible to set up with the information available at this stage. In our example, this means that the protein `protein tetRdimer` is initialized, so that this version of the TetR dimer is used in the specification of the repression reaction, which happens in the second, reaction mode call to `txtl_add_dna` by the `txtl_runsim` command.

One idea hinted at in the above discussion is that when the species are being set up, there might not even be enough information available to set up all the species required. Some species appear as the products of reactions, and are only known once the reactions are specified. Indeed, if species-version dependent reactions lead to product species whose exact version other reactions depend on in turn, then the above two pass method will not suffice. Though we do not implement the solution in this version of the toolbox, one can imagine a multi-pass architecture that alternates between calls to `txtl_add_dna` in species setup and reaction setup modes, with the iterations ending only when the set of reactions and species no longer grows.

In both modes of the call to txtl_add_dna, the command performs the following actions: call the component function files for the promoter, the UTR and the CDS, followed by a function to set up mRNA degradation species and reactions, followed by DNA and protein degradation, if present. The promoter file sets up reactions and species (depending on the mode) associated with TF mediated regulation and transcription. Similarly, the UTR function file sets up ribosome binding reactions and other reactions associated with translation.

Returning to the user level code, once all the txtl_add_dna commands have been specified, the extract, buffer and DNA model objects (with variable names starting with tube) are combined in using the txtl_combine command, which simply adds the species and reactions from the three model objects into a single model object, and scales the concentrations of the species to simulate the resulting change in volume. The resulting model object, often named as a variable Mobj, can be simulated by txtl_runsim. Note that even if simulation is not the immediate goal, one call to txtl_runsim should always be performed, since this is where the reactions in the model are set up with the txtl_add_dna command. After the call to txtl_runsim, we have a fully defined model object, and a simData class object containing the results of the simulation. These objects may be used for further simulations, parameter inference, and visualization of the species trajectories, of be exported to other platforms via SBML.

3.5 Tools for Multi-Experiment Concurrent Bayesian Parameter Inference

Bayesian parameter inference via MCMC methods involves designing a reversible Markov chain with stationary distribution matching the posterior parameter distribution (given models and data). This Markov chain can then be simulated and sampled to build an ensemble of points that estimates the desired parameter distribution. One example of this is the Metropolis-Hastings sampler, which was used for parameter inference in [11]. Numerous variations and extensions of MCMC samplers exist, and we use the ensemble sampler by Goodman and Weare [25], which is particularly well suited to highly anisotropic densities that occur due to parameter non-identifiability in biological models.

In this section we present mcmc_simbio, which performs concurrent Bayesian parameter inference on Simbiology® models. By concurrent parameter inference, we mean the following. Suppose we have a set of different experiments, with a model corresponding to each experiment. Let each experiment-model pair be used to estimate some set of parameters, with the possibility that parameters may be informed by more than one model-experiment pair. Concurrent parameter inference finds the posterior distribution for the parameters given the full set of experiment-model pairs and the specification of the subset of parameters informed by each experiment. This scheme is depicted visually in Figure 3.8. mcmc_simbio builds the concurrent estimation capabilities and Simbiology® specific features around the MATLAB® implementation of the Goodman and Weare ensemble MCMC sampler [20, 25, 28].

One application of this toolbox is during the calibration step of the calibration-correction method introduced in Chapter 2. The calibration step of the method involves sharing the circuit specific parameters (CSPs) between two extracts (i.e., estimating a single set of values for them) while estimating individual sets of values for the extract specific parameters (ESPs). Recall from Section 2.4.2 that we fit the calibration data for each extract in Figure 2.4 to a corresponding model, with each CSP point (comprising the sole parameter coordinate k_{rG}) in the ensemble estimated to fit both models to their respective data sets simultaneously, while each model-data pair fits its own ESPs (Enz and k_c) independently of the other. This scheme is summarized in Figure 3.9 (A), where each dot represents the set of ESPs or CSPs for one model-experiment pair (determined by the circuit and extract used). A line between two dots indicates that if a parameter appears in both the sets represented by the dots, then it is estimated jointly or concurrently. Figure 3.9 (B) shows a different sharing pattern, where the CSPs are shared between the extracts for both the calibration circuit and the test circuit, and the ESPs are shared between the circuits for each extract. This, and other variations to this pattern, might be useful for comparing with the sets of parameters obtained by the base calibration-correction method.

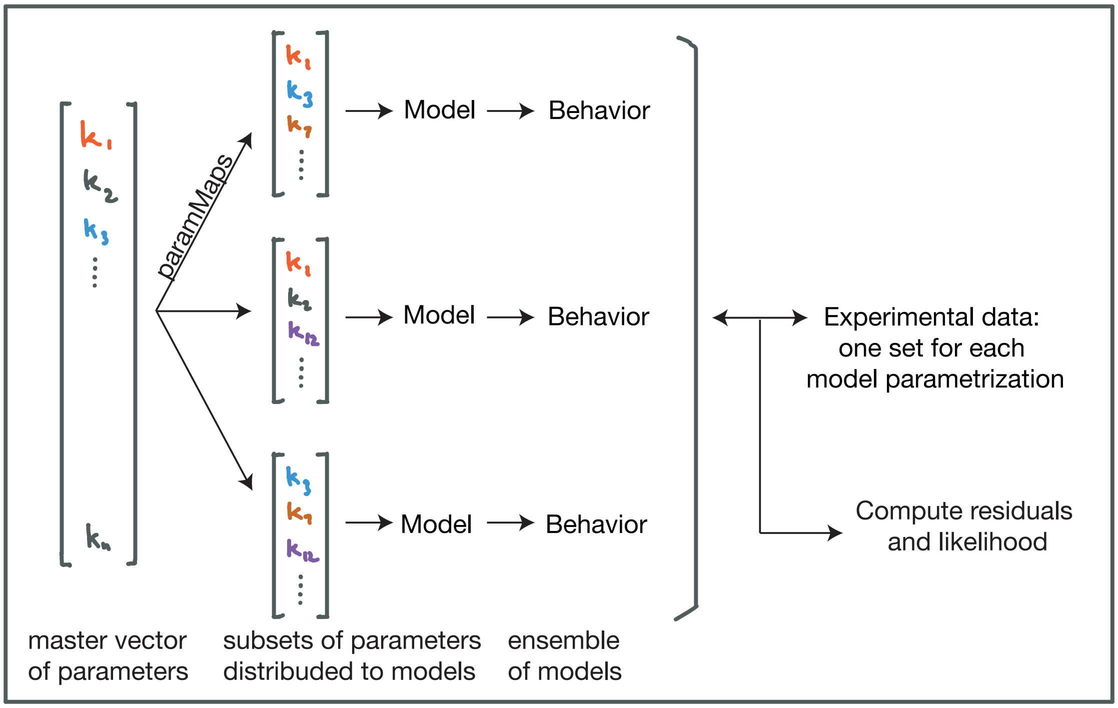

Figure 3.8: Parameter sharing in setting up the concurrent parameter inference problem. Given different sets of experimental data, and corresponding models, which can differ in the structure of the chemical reaction network (network 'topology') or just the parameter values in the models (network 'geometry'), the concurrent parameter inference problem is set up as follows. A master vector is defined, which collects all the parameters in the models into a single vector. Each parameter that is to be shared between models only appears once in the master vector. I.e., parameters that are to be identified with each other between models are treated as an equivalence class of parameters, and their representative is placed in the master vector. Next, paramMaps matrices (described in Appendix 3.C) are used to distribute the parameters to models, which are then simulated and their behavior compared against corresponding experimental data to compute the likelihood values for the purposes of the Bayesian Parameter inference.

The ESPs are shared between models of different circuits, corresponding to different biochemical network topologies, while the CSPs are shared between models that differ only in their parameter values. We refer to the first type of sharing as sharing between model *topologies* while the latter as parameter sharing between model *geometries*. In general, each model can be specified by a unique pair of indices, the first of which specifies the model's topology, and the second the model's geometry. Thus, we will often refer to models as topology-geometry pairs.

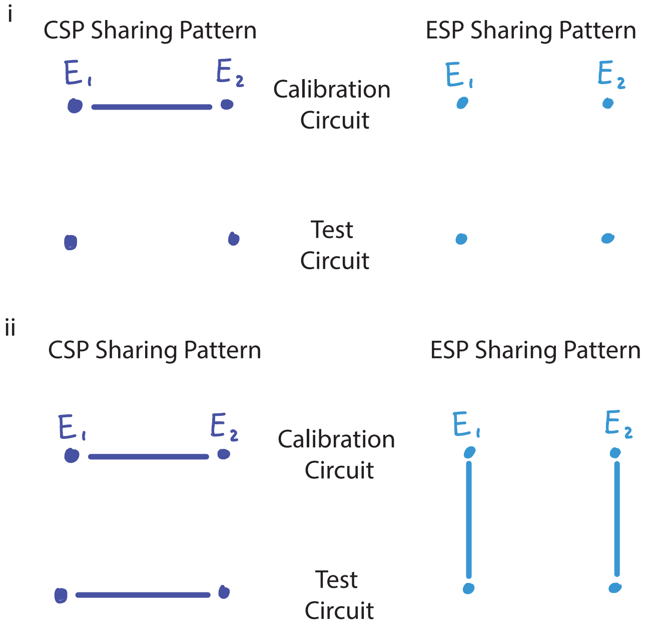

Figure 3.9: Application of the concurrent Bayesian inference capabilities to the calibration-correction problem of Chapter 2. (i) The sharing pattern for the calibration-correction method. At the calibration step, only a single set of values for the circuit specific parameters is estimated. there is no other sharing present. (ii) A sharing pattern where circuit specific parameters are shared within a single model topology (between geometries) and extract specific parameters are shared between circuits within a single extract. We are using different sharing patterns like this to explore, derive and verify the types of mathematical conditions derived in Sections 2.5 and 2.6 of Chapter 2.

An advantage of estimating the entire joint posterior distribution of the parameter, as opposed to using optimization methods for point estimation, is that it can be used to check the identifiability properties of the models. This is useful for understanding which parameters are well constrained by the data, if there is any covariation present between the parameter estimates (Section 2.6), and for designing experiments for reducing parameter non-identifiability. Indeed, a particularly useful application of concurrent parameter inference is in experiment design to reduce or remove parameter non-identifiability. One can iterate on the set of models and data, possibly of heterogeneous forms, to find the smallest set that gives identifiable parameters. Indeed, the experiments do not even have to be performed at the design stage, and models may be used to generate artificial data from each model, from which parameter identifiability can be checked.

3.5.1 An Illustrative Example

In this section, we describe the concurrent parameter inference capabilities of mcmc_simbio in some detail using an example similar to the calibration step in the calibration-correction method. Recall from Section 2.4.3 that the calibration step for the example in Chapter 2 involved a model given by

$$\mathrm{D_G} + \mathrm{Enz} \underset{k_{rG}}{\overset{k_{fG}}{\rightleftharpoons}} \mathrm{D_G{:}Enz} \xrightarrow{k_c} \mathrm{D_G} + \mathrm{Enz} + \mathrm{G}, \tag{3.5}$$

where $\mathrm{D_G}$ is the GFP DNA, Enz is the enzyme used to model the transcriptional and translational machinery, $\mathrm{D_G{:}Enz}$ is a complex, and G is the GFP protein. The reaction rate parameters are k_{fG}, k_{rG}, and k_c respectively. The calibration step requires the implementation of this circuit in two extracts, and in the language of mcmc_simbio, we say that there is one topology (circuit, or network topology) and two geometries (implementations of that circuit in different extracts, differing only in their parameter values), leading to two topology-geometry pairs (models). Together, the two models have to be fit to corresponding data sets to estimate an ensemble of parameter values.

The experimental data associated with this parameter inference problem involve the implementation of this circuit in two extracts, at three different initial DNA concentrations ('doses' in the language of mcmc_simbio), with time courses of GFP measured ('measured species'). The parameters expected to be the same across the two extracts are those that pertain to the circuit parts, i.e., the binding-unbinding rates k_{fG} and k_{rG}. In our estimation problem, we set the value of k_{fG} to its true value (the value used to generate the artificial data), and only estimate k_{rG} jointly. The parameters to be estimated individually for each model are those expected to be different between the two extracts, i.e., the initial enzyme $[\mathrm{Enz}]_0$ concentration and the elongation rate k_c. The resulting parameter space being searched is five dimensional ($\theta = (k_{rG}, [\mathrm{Enz}]_{1,0}, k_{c1}, [\mathrm{Enz}]_{2,0}, k_{c2})$). Figure 3.10 shows the result of estimating the ensemble of parameter points (see Figure 3.10C, D for pairwise projections of the log transformed values of the ensemble) that fit the simulated data to the models. We picked fifty points from the estimated ensemble, and generated model prediction trajectories for each DNA dose (initial condition) and in each of the two extracts (Figure 3.10A, B). Figure 3.11 shows the Markov chains obtained by performing this MCMC run.

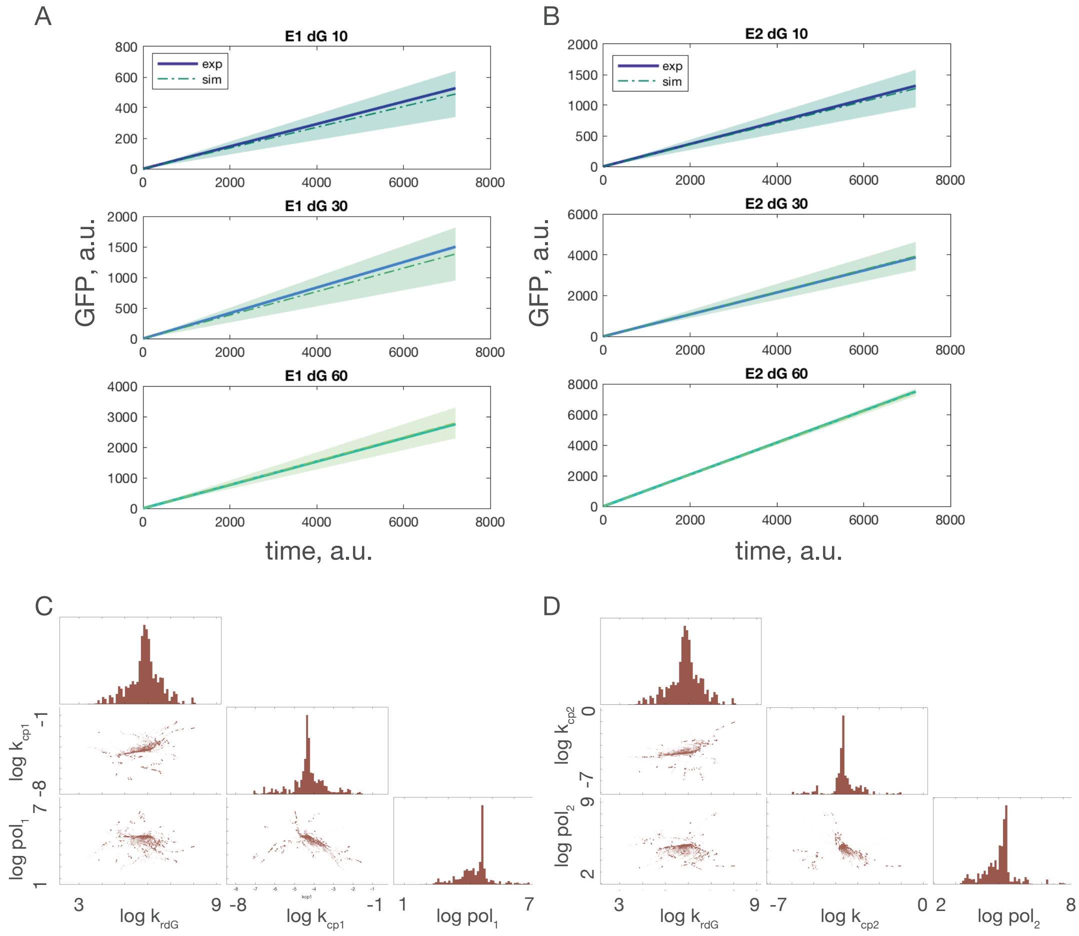

Figure 3.10: mcmc_simbio example. (A, B) Model fits to artificially generated data. Solid line: artificially generated experimental data. Dashed line: mean of 50 simulated trajectories resulting from the ensemble of parameter estimates. Shaded region: standard deviation. (C, D) Pairwise projections of the posterior parameter distributions.

The setup of this estimation problem involves setting up a `proj_<projname>.m` project file,

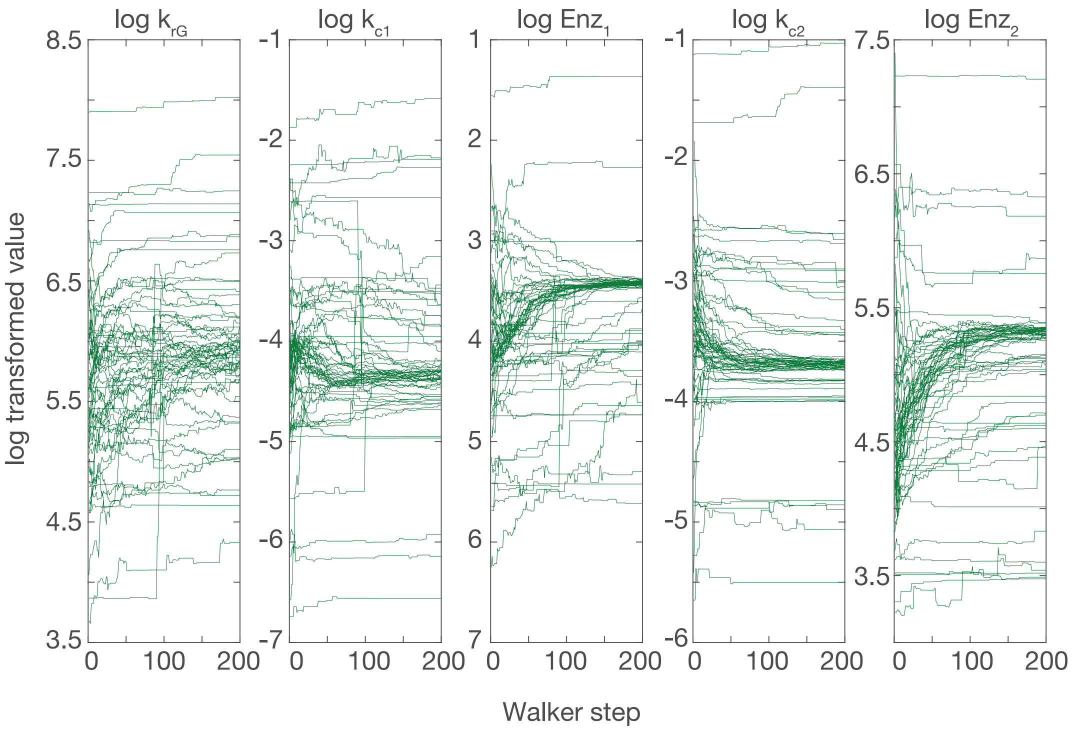

Figure 3.11: Markov chains in the mcmc_simbio example.

which contains information on the experimental data, Simbiology models, specifications of the parameter sharing pattern, the hyperparameters for the MCMC algorithm, and the data visualization specifications. The general layout of the file for this example is shown in the code below.

```
% Initialize the project directory, where the data and plots will be stored.
[tstamp, projdir, st] = project_init;

% Define the simbiology model class object to be used. In this problem there is
    only one topology (circuit): the constitutive gene expression circuit.
    model_protein3 is a file that sets up the appropriate model.
mobj = model_protein3;

% define the mcmc_info struct that specifies the estimation problem structure and
    hyperparameters. See detailed discussion below describing this struct.
mcmc_info = mcmc_info_constgfp3ii(mobj);
```

```
% The model_info field in the mcmc_info struct is a MALTAB struct in itself, and
    contains information about the model topologies, geometries and parameter
    concurrence pattern.
mi = mcmc_info.model_info;

% A list of nominal parameter values to use to generate the data.
rkfG = 5; rkrG = 300; rkc1 = 0.012;
rkc2 = 0.024; cEnz1 = 100; cEnz2 = 200;

% Arrange the parameters in a log transformed 'master' vector.
masterVector = log([rkfG; rkrG; rkc1; rkc2; cEnz1; cEnz2]);

% Generate artificial data for the two extracts using the model object, a vector
    of timepoints, the set of parameters, and information of which species are to
    be dosed and measured.
di = data_artificial_v2({mobj}, {0:180:7200}, {mi.measuredSpecies},...
   {mi.dosedNames}, {mi.dosedVals}, {mi.namesUnord},...
    {exp(masterVector([1:2 3 5])), exp(masterVector([1:2 4 6]))});

% perform the ensemble MCMC parameter estimation.
mi = mcmc_runsim_v2(tstamp, projdir, di, mcmc_info);

% Plotting commands
% get the mcmc chains from saved timestamped data.
marray = mcmc_get_walkers({tstamptouse}, {1:ri.nIter}, projdir);

% plot the parameter distribution corner plots and markov chains
mcmc_plot(marray, mai.estNames, 'tstamp', tstamptouse);

% plot the data trajectories and the simulated data fits.
mvarray = masterVecArray(marray, mai);
marrayOrd = mvarray(mi(1).paramMaps(mi(1).orderingIx, 1),:,:);
fhandle = mcmc_trajectories(mi(1).emo, di(1), mi(1), marrayOrd,...
   titls, lgds, 'projdir', projdir, 'tstamp', tstamptouse, 'extrafignamestring',
       '_extract1');
```

```
% and more plotting commands may be added as needed...
```

The command `mcmc_info_constgfp3ii` is used to set up a MATLAB® 'struct' class object called `mcmc_info`. This struct has three fields, `model_info`, `runsim_info` and `master_info`, which are themselves MATLAB® structs.

The `model_info` struct array, having one entry for each topology, is used to specify information about the models used in the estimation problem. This information includes the full list of parameters in each model topology (`namesUnord`), a specification of how the parameters in the `masterVector` field of the `master_info` struct are to be distributed to each model, and information on dosing (initial conditions) and measurement (output) for each model. This struct is described in detail in Appendix 3.C.

The `master_info` struct is used to specify information about the pool of parameters to be shared across all the topology-geometry pairs. Along with the `masterVector`, it also contains the fields `estNames`, `paramRanges`, and `fixedParams`, which are described below.

Finally, the `runsim_info` struct is used to specify the simulation hyperparameters like the number of points to simulate the chains for, the noise model, the number of MCMC 'walkers' (chains), the step size for the algorithm, whether parallelization is to be used, etc.

Next, we outline how these structs are used to set up the estimation problem. We set up parameter concurrence by first specifying a vector of parameters called the `masterVector`. This vector contains all the parameter values that are to be distributed to all of the topology-geometry pairs. We allow values within the `masterVector` to be either *fixed* or *estimated*. The `masterVector` is initialized to a set of values in the file `mcmc_info_constgfp3ii.m`, and the `master_info.fixedParams` field is used to specify which of these values is to be fixed. The remaining values constitute the vector of parameters to be estimated during the MCMC process, and are named by the cell array `master_info.estNames`. At each iteration of the algorithm, MCMC generates a new proposal of the estimated parameter vector. This proposal is used to populate the relevant entries in the `masterVector`, and the `paramMaps` field of the `model_info` struct is used to distribute the parameters from the `masterVector` to the individual model geometries. Each model is then simulated at each of the dosing conditions (specified by the dosedNames and dosedVals fields of the model_info struct), and the data for the species to be measured are compared to the experimental data stored in the data_info struct array. The dataToMapTo and measuredSpeciesIndex fields in model_info are used to specify which element of the data_info struct array a given model's output corresponds to, and the mapping from the model's species to the experimental data trajectories in the data set.

For our example, the code snippets below show the section of mcmc_info_constgfp3ii.m that are used to specify this functionality. The comments, shown in green, are used to link the description above to specific functionalities. First, we show the top level constituents of the mcmc_info struct.

```
% In this example, model_info is a scalar struct, since there is only one
    topology. In general, each topology gets its own element in this struct.
model_info = struct(...
    'circuitInfo',{circuitInfo},...
    'modelObj', {modelObj},...
    'modelName', {modelObj.name},...
    'namesUnord', {namesUnord}, ...
    'paramMaps', {paramMap}, ...
    'dosedNames', {dosedNames},...
    'dosedVals', {dosedVals},...
    'measuredSpecies', {measuredSpecies}, ...
    'measuredSpeciesIndex', {msIx},...
    'dataToMapTo', dataIndices);

% The master_info struct is a scalar struct and gives the initial masterVector of
    parameter values to be distributed to the topology geometry pairs, which of
    the indices in that vector are to be fixed (fixedParams vector of indices), a
    string of names of parameters (and species initial concentrations) that are
    to be estimated (estParams), and the range of values to seach over for each
    parameter.
master_info = struct(...
    'estNames', {estParams},...
    'masterVector', {masterVector},...
    'paramRanges', {paramRanges},...
    'fixedParams', {fixedParams});
```

The next code snippet describes how each of the entries of the `model_info` and `master_info` structs is specified. For the single topology in this example, there are two geometries. This is encoded by the fact that the `paramMaps` field of the `model_info` struct is a matrix with two columns, as shown in the code snippet below.

```
% Information describing the circuit. This gets printed in the log file. Here,
    the enzymatic reaction is used to produce the protein G. Since there is only
    one topology, only one string is needed.
circuitInfo = ...
    [' D_G + Enz <-> D_G:Enz (kfG, krG \n'... )
    'D_G:Enz -> G + Enz + protien (kc)\n'...
    'single topology, two geometries.'];

% The masterVector of all the paramters: both fixed and estimated. This vector is
    used during the MCMC algorithm.
% The fixed parameter (kfG) is fixed at a value of 5 (arbitrary units) here, and
    its index in the masterVector is specified by fixedParams.
% At each iteration of the MCMC algorithm, a new 5D parameter point is proposed,
    and used to update
% the relevant entries of the master vector. The values in this vector are
% then distributed to the two geometries.
rkfG = 5; rkrG = 300; rkc1 = 0.012; rkc2 = 0.024; cEnz1 = 100; cEnz2 = 200;
% Note that the values in the masterVector are log transformed.
masterVector = log([rkfG; rkrG; rkc1; rkc2; cEnz1; cEnz2]);
% just the rkfG parameter is fixed, which has index 1 in masterVector
fixedParams = [1];
% The remaining indices are the estimated parameters. The indices are [2:6]
estParamsIx = setdiff((1:length(masterVector))', fixedParams);

% namesUnord is a list of the species and parameters in the model that are set
    from values drawn from the masterVector. These incluce both the fixed and
    estimated values. In each model, we have the parameters 'kfG', 'krG', and
    'kc' whose values get set and the species 'Enz' whose initial value gets set.
namesUnord = {'kfG';'krG';'kc';'Enz'};
```

```
% estParams is a cell array of strings containing the names of the species and
    parameters in the masterVector that are not fixed. There are five values
    here: krG, which is estimated jointly for both geometries, and kc and Enz,
    each of which are estimated separately for each geometry (labeled 1 and 2).
estParams = {'krG';'kc1';'kc2';'Enz1';'Enz2'};

% The paramMaps field is a matrix that maps the elements of the masterVector to
    the individual parameters and species in the topology-geometry pairs. For a
    given topology, we have one matrix, with the number of columns specifying the
    number of geometries associated with that topology, and how the parameters
    from the master vector are to be distributed to each geometry. In this case,
    there are two geometries: the first geometry's parameters and species,
    specified by namesUnord ('kfG', 'krG', 'kc' and 'Enz'), are set to be
    specified (during each MCMC iteration) by indices 1, 2, 3, and 5 of the
    masterVector, i.e., kfG, krG, kc1 and Enz1. Similarly, the second geometry's
    namesUnord species and parameters are set to be specified by
    masterVector(mcmc_info.model_info(1).paramMaps(:,2)), i.e., kfG, krG, kc2,
    and Enz2.
paramMap1 = [1 2 3 5]';
paramMap2 = [1 2 4 6]';
paramMaps = [paramMap1 paramMap2];

% paramRanges: A length(masterVector) by 2 matrix of the ranges of (log
    transformed) values to limit the MCMC sampling to. We limit the search in
    this example to +-3 from the values used to generate the artificial date.
paramRanges = [masterVector(estParamsIx)-3 masterVector(estParamsIx)+3];

% The data_info struct array contains the data sets associated with this
    estimation problem. In this problem, this array is of length two, with the
    first struct entry corresponding to the first geometry, and the second struct
    entry corresponsing to the second geometry.
dataIndices = [1 2];

% next we define the dosing strategy. The species names dG in the Simbiology
    model is to be dosed, and at the values specified.
```

```
dosedNames = {'dG'};
dosedVals = [10 30 60];

% define the species to be measused. Here the species named pG is measured.
measuredSpecies = {{'pG'}};

% The trajectories of the pG species get mapped to the column with index msIx = 1
    in the data_info(dataIndices(i)).dataArray matrix, where i is a geometry
    index.
msIx = 1; %
```

After the `mcmc_info` struct has been defined, the `data_info` struct array is specified. In this example, known models are used to generate artificial data, but in general this struct is defined using real experimental data. In general, `data_info` is a struct array. The (i, j)-th topology-geometry pair uses data specified in

```
data_info(mcmc_info.model_info(i).dataIndices(j)).
```

The struct is used to specify a vector of time points, a list of names of species that are measured, a list of names of species that are dosed, a matrix of dose values, a four dimensional array of data values, and other metadata. This is summarized in Table 3.C.1 in Appendix 3.C.

Once these structs have been defined, they are used as inputs into the `mcmc_runsim` function, which performs the concurrent parameter inference, and saves the results and log files in a time-stamped subdirectory within the toolbox. The toolbox also contains plotting functionalities, functionality for generating data_info structs populated with artificial data, and for converting raw platereader data into data_info structs.

3.6 Discussion

In this chapter, we have described `txtlsim`, a toolbox for simulating batch mode TX-TL reactions using Simbiology®, and mcmc_simbio, a smaller toolbox within `txtlsim` that performs concurrent Bayesian parameter inference on Simbiology® models (not just `txtlsim` models). The key features of txtlsim are that it requires only a few lines of code to generate a model of gene regulatory circuits within TX-TL with enough complexity to model the loading of transcription, translation and RNAse catalytic machinery, and the consumption of resources like nucleotides and amino acids. The requirement for modeling resource consumption while keeping the reaction network size manageable led to the creation of consumption reactions with reaction rates defined to be a function of polymer length and mRNA or protein production rates. These reactions are discussed in greater depth in Chapter 4. The txtlsim toolbox also provides support for a wide range of regulatory parts, and is easily extensible by users. Furthermore, the modeling framework of txtlsim automatically accounts for retroactivity and loading effects, without needing for these to be explicitly specified in the model equations. We have described the usage of txtlsim, and the software architecture needed to automatically generate a complex chemical reaction network from simply specified user inputs. We have validated the model by characterizing core and part parameters using data from the literature, and from experiments performed in the lab, and predicting the behavior of an incoherent feedforward loop circuit.

The mcmc_simbio toolbox enables for different sets of experiments, possibly from heterogeneous sources, to be combined for parameter inference purposes, allowing for more information to be incorporated into the parameter inference problem. Indeed, since the approach returns the joint posterior parameter density, the improvements in parameter identifiability resulting from using multiple experiments to estimate parameters can be checked visually. While we do not show the use of this toolbox for inferring txtlsim parameters in this chapter, we do use the toolbox for parameter inference performed in Chapter 2.

There are numerous directions that this work may be extended in. Firstly, capabilities from the MATLAB® based GenSSI toolbox [10] for checking structural identifiability of experiments-model pairs may be added to txtlsim. GenSSI uses Lie derivatives of the model output with respect to the parameters to generate approximations to the so called exhaustive summary of model parameters given the initial conditions and outputs of the model. The exhaustive summary contains all the information that can be learned about the parameters, and if the map from the parameters to the exhaustive summary is injective, the parameters can be shown to be identifiable in the sense of Definition 2. Using GenSSI, along with Bayesian inference on artificial `txtlsim` data, to explore identifiability would form a potent approach for model checking and experiment design.

Another extension of this work would be the incorporation of 'modes' of simulation within `txtlsim`. We might choose to turn on or off reactions to model growth and dilution as part of a 'cell' or 'microfluidics' mode. We may also include modes for more or less detailed models, such as lumping transcription and translation into single reactions, or switching to Hill kinetics from mass action kinetics.

Other extensions include the ability to port models to the bioscrape toolbox [62] and for the models generated by `txtlsim` and other tools to be treated as semantically distinct elements, and be interconnected as subsystems into a larger system.

All in all, we believe that if modeling based approaches are flexible, easy to use and biologically faithful enough for the modeling purpose they are intended for, then they will actually be used by the synthetic biology practitioner, and help accelerate the progress of the field. Our hope is that `txtlsim`, mcmc_simbio, and their extensions help advance this vision.

Appendices

3.A Consumption Reactions as a Means of Tracking Resource Utilization in Reduced Models of Transcription and Translation

In this section, we discuss the use of consumption reactions to maintain the correct stoichiometry of resource utilization during transcription and translation, while still allowing for detailed elongation models to be replaced by single step reactions. An in depth discussion of this subject may be found in Chapter 4 .

Consider the transcription of an mRNA species of length 1kb. Assume that the four types of bases are equally distributed along the mRNA, and so 250 molecules each of ATP, GTP, CTP and UTP are required for the transcription of this mRNA species. In our model, ATP and GTP are modeled together as a species AGTP, where we assume that one unit of the AGTP represents one unit of ATP and one unit of CTP. Similarly, one unit of CUTP represents one unit of CTP and one of UTP. Thus, 250 units each of AGTP and CUTP are needed to transcribe the 1kb mRNA molecule. Looking at the model in Equations (3.1), we see that the mRNA production step reaction consumes one unit each of AGTP and CUTP and produces one mRNA molecule. The consumption reaction also consumes one unit each of AGTP and CUTP, and does not produce an mRNA molecule. Thus, to consume 250 units each of AGTP and CUTP per mRNA produced, we may set the rate of the consumption reaction to be $L_m/4-1=249$ times the rate of the mRNA production step. We now show that with this choice, the correct number of nucleotides gets used per mRNA molecule produced. The rate of mRNA production is given by

$$\frac{\mathrm{d[mRNA]}}{\mathrm{d}t}=k_{tx}\cdot[\mathrm{CUTP{:}AGTP{:}RNAP{:}DNA}].$$

To compute the rate of nucleotide consumption, we define a variable N_{uninc}, which is the total concentration of nucleotides not incorporated into mRNA. Ie, $N_{\text{uninc}} = 4\cdot[\text{CUTP:AGTP:RNAP:DNA}] + 2\cdot([\text{AGTP:RNAP:DNA}] + [\text{CUTP:RNAP:DNA}] + [\text{CUTP}] + [\text{AGTP}])$. We would like to show that the rate at which these unincorporated nucleotides are decreasing is $L_m = 1000$ times the rate at which the mRNA is being produced. The rate of consumption of unincorporated nucleotides is calculated as

$$\begin{aligned}
\frac{\mathrm{d}N_{\text{uninc}}}{\mathrm{d}t} &= 4\cdot\frac{\mathrm{d}\left([\text{CUTP:AGTP:RNAP:DNA}]\right)}{\mathrm{d}t}, \\
&\quad + 2\cdot\left(\frac{\mathrm{d}[\text{AGTP:RNAP:DNA}]}{\mathrm{d}t} + \frac{\mathrm{d}[\text{CUTP:RNAP:DNA}]}{\mathrm{d}t} + \frac{\mathrm{d}[\text{CUTP}]}{\mathrm{d}t} + \frac{\mathrm{d}[\text{AGTP}]}{\mathrm{d}t}\right), \\
&= -4\cdot\left(k_{tx} + \left(\frac{L_m}{4} - 1\right)k_{tx}\right), \\
&= -L_m\cdot k_{tx},
\end{aligned}$$

where the second equality follows from converting Equations (3.1) into the corresponding mass action ODEs and substituting these into the derivative terms above, and observing that most of the terms in the resulting expression cancel in pairs. We note that the derivation of the consumption reactions for translation is exactly analogous, and the only thing that needs to be stated is that on average, the energetic cost of translation involves four ATP equivalents (two ATP and two GTP) per amino acid incorporation.

3.B MATLAB® Simbiology®

The MATLAB® Simbiology® toolbox follows the SBML standard in its class structure, with classes for models, compartments, species, reactions, parameters, rules, events, kinetic laws and other features. At the top level we have a Simbiology® *model* class object that contains one or more *compartment* class objects. To each compartment, one may associate reaction, species, rule, event, parameter and kinetic law class objects. Individual kinetic law objects, which are associated to a unique parent reaction, are used to specify the reaction properties like the reaction rate law and parameters associated to that reaction. The parameters within a kinetic law refer to parameter class objects, which can be scoped either at the model level or the kinetic law levels. Parameter objects scoped at the model level can be used by multiple kinetic law objects, while those scoped within a kinetic law object can only be used by that object. Species objects can form either the reactants or products of a reaction, and are scoped at the compartment level. Rules are relationships between parameters, rates and species, and events allow the modeling of discontinuous dynamic changes in the model.

3.C Details of the Data Structures used to Specify the Concurrent Parameter Inference Problem

The `data_info` struct is a MATLAB® struct class array of length nDataSets, where nDataSets is the number of data sets used in the parameter inference problem. Table 3.C.1 gives descriptions of the contents of each field for each element within this struct.

Table 3.C.1: The fields of the `data_info` struct.

Field	Description
`dataInfo`	A human readable description of the data.
`timeVector`	A vector of timepoints of length nTimePoints.
`timeUnits`	A string specifying the time units. Most commonly 'seconds', 'minutes' or 'hours'.
`dataArray`	4-D array of data of size nTimePoints by nMeasuresSpecies by nReplicates by nDoseCombinations.
`measuredNames`	An array of strings representing the names of the measured species. It has length nMeasuredSpecies.
`dataUnits`	An array of strings specifying the units each measured species was measured in. It has length nMeasuresSpecies
`dosedNames`	An array of strings representing the names of the dosed species. It has length nDosedSpecies.
`dosedVals`	A matrix of dose values, of size nDosedSpecies by nDoseCombinations.
`doseUnits`	An array of strings specifying the units of each of the dosed species. It has length nDosedSpecies.

Similarly, the `model_info` struct is of length nTopologies, where nTopologies is the number of different models used in the parameter inference problem. Table 3.C.2 gives descriptions of the contents of each field within this struct for each element within the struct array.

Table 3.C.2: The fields of the `model_info` struct. This struct is of length nTopologies, and specifies the properties of models, and the pattern of parameter sharing across the topologies and geometries for the purposes of setting up the concurrent parameter inference problem.

Field	Description
`circuitInfo`	A human readable description of the model.
`modelObj`	A Simbiology® model class object (in the terminology of the concurrent parameter inference problem, this is a network *topology*).
`namesUnord`	A list of parameters in the model object that are set from values in the master_vector.
`paramMaps`	A matrix of the indices of the `master_vector` that correspond to the parameters specified in the list `namesUnord`. Each column of this matrix specifies one set of elements of the `master_vector` that specify the values of the parameters in `namesUnord` for this model. The number of columns, nCols, of this matrix is the number of different *geometries* of the model, in that the models are different, but only in the values the parameters take, and not in the network topologies.
`dosedNames`	An array of strings representing the names of the dosed species. It has length nDosedSpecies.
`dosedVals`	A matrix of dose values, of size nDosedSpecies by nDoseCombinations.
`measuredNames`	An array of strings representing the names of the measured species. It has length nMeasuresSpecies.
`measuredSpeciesIndex`	An array of indices pointing to the measured species columns of the `dataArray` in the `data_info` struct.
`dataToMapTo`	A numerical vector of length nCols containing the indices of the elements of the `data_info` struct that the model geometries correspond to. These are used when the model predictions are compared to the data in the computation of the log likelihood during MCMC.

The function `mcmc_runsim` generates an inference problem as follows. Suppose there are nTopologies different model topologies specified by `model_info`. Let the topologies be indexed by the letter i. Suppose that for the i-th topology, the corresponding `paramMaps` matrix has nCols_i columns, each corresponding to a geometry. Then, `mcmc_runsim` creates an ensemble of nCols_1 $+ \cdots +$ nCols_nTopologies models, and uses the `paramMaps` matrices to distribute the parameter values in `master_vector` into this ensemble of models. All of these models are then simulated, the residuals generated by comparing the results to the `data_info` elements specified by the `dataToMapTo` field, and the log likelihood computed. The MCMC algorithm uses this to compute the new points in the space of estimated parameters and updates the `master_vector` with the new proposals. The algorithm then repeats until a stopping criterion, such as the number of points to simulate the Markov chains for, is met.