VIVARIUM: AN INTERFACE AND ENGINE FOR INTEGRATIVE MULTISCALE MODELING IN COMPUTATIONAL BIOLOGY

Eran Agmon
Department of Bioengineering
Stanford University
Stanford, CA, USA
agmon.eran@gmail.com

Ryan K. Spangler
Allen Institute for Cell Science
Seattle, WA, USA

Christopher J. Skalnik
Department of Bioengineering
Stanford University
Stanford, CA, USA

William Poole
Computation and Neural Systems
California Institute of Technology
Pasadena, CA, USA

Shayn M. Peirce
Department of Biomedical Engineering
University of Virginia
Charlottesville, VA, USA

Jerry H. Morrison
Department of Bioengineering
Stanford University
Stanford, CA, USA

Markus W. Covert
Department of Bioengineering
Stanford University
Stanford, CA, USA
mcovert@stanford.edu

April 27, 2021

ABSTRACT

Motivation: This paper introduces Vivarium – software born of the idea that it should be as easy as possible for computational biologists to define any imaginable mechanistic model, combine it with existing models, and execute them together as an integrated multiscale model. Integrative multiscale modeling confronts the complexity of molecular and cellular biology by combining heterogeneous datasets and diverse mechanistic modeling strategies into unified representations. These integrated models are then run to simulate how the hypothesized mechanisms operate as a whole. But building such models has been a labor-intensive process that requires many contributors, and they are still primarily developed on a case-by-case basis with each project starting anew. New software tools that streamline the integrative modeling effort and facilitate collaboration are therefore essential for future computational biologists.
Results: Vivarium is a software tool for building integrative multiscale models. It provides an interface that can make any mechanistic model into a module that can be wired together into larger composite models and then parallelized and run across multiple CPUs with Vivarium's simulation engine. The utility of this software is demonstrated by building multi-paradigm composite models that combine several popular modeling frameworks: agent based models, ordinary differential equations, stochastic reaction systems, constraint-based models, solid-body physics, and spatial diffusion. This demonstration shows just the beginning of what is possible – future efforts can integrate many more types of models and at many more biological scales.
Availability: The simulations and output used for this paper are available as Jupyter notebooks at `https://github.com/vivarium-collective/vivarium-notebooks`. The vivarium-core multiscale engine has been released as a PyPI library and can be installed with `pip install vivarium-core`. Additionally, vivarium-core is open-sourced for development at `https://github.com/vivarium-collective/vivarium-core`. Vivarium libraries used for this paper are listed in the Supplementary materials.

Keywords Integrative modeling · Multiscale modeling · Computational biology

1 Introduction

Our understanding of biological phenomena stands to be dramatically improved if we can adequately represent the underlying systems, mechanisms, and interactions that influence their behavior over time. Generating these representations, most commonly via mathematical and computational modeling, is made challenging by the complex nature of such systems. We have seen a proliferation of diverse observational data on the molecular composition, spatial organization, and dynamics of thousands of cell types [1]. The most common modeling approaches today that use such data sets are statistical, extracting meaning by fitting functions such as regression, cluster analysis, and neural networks. Although these models have been successful at approximating correlations among observed variables, the structures of the models are not easily interpretable, making it difficult to derive biological mechanism from their predictions. In contrast, mechanistic models are designed to reproduce observed data by representing causality [2]. Thus, the mathematical form and parameters of mechanistic models are testable hypotheses about the system's underlying interactions. In other words, mechanistic models can provide unique insights that can confirm or refute hypotheses, suggest new experiments, and identify refinements to the models. Some exciting progress has recently been made in combining both strategies [3, 4].

Mechanistic models in computational biology have deepened our understanding of diverse domains of biological function, from the macromolecular structure and dynamics of a bacterial cytoplasm with atomistic models [5], the lysis/lysogeny switch of bacteriophage lambda with a stochastic kinetic model [6], bacterial growth in different conditions with constraint-based models of metabolism [7], and cell-based models of quorum sensing in bacterial populations [8]. However, such models generally target a mechanism in isolation, with a single class of mathematical representation, and focus on a narrow range of resulting behavior. A logical next step in the development of computational biology is to combine these components and build upon their insights, so we can better understand how their represented mechanisms operate together as integrated wholes. Organisms are fundamentally multi-modal and multiscale, driven by mechanisms ranging from individual molecular binding events or conformational changes, to the growth and development of ecosystems over evolutionary time. To accurately represent such systems, our models must also be multi-modal and multiscale.

Integrative models combine diverse mechanistic representations with heterogeneous data to represent the complexity of biological systems. There have been several such efforts including the integrative modeling of whole-cells [9, 10], macro-molecular assemblies [11], microbial populations [12], and even some work towards whole-organisms [13]. They have shown some success in capturing the emergence of complex phenotypes – but many challenges remain to the extensibility of the resulting models and to their widespread adoption. This results in a loss of research momentum and an apparent ceiling on model complexity. The ideal model would be not only be integrative in terms of incorporating diverse mathematical approaches and biological functions, but also in terms of bringing together the vast scientific expertise across the globe. What is therefore required is a methodology that brings molecules and equations, as well as labs and scientists, together in this effort.

In this regard, software infrastructure can greatly facilitate the development of integrative models. Two major areas of development in this space are standard formats and modeling frameworks. Standard formats allow models to be shared and distributed between different software tools – just as HTML allows web pages to be viewed across multiple browsers and devices. Popular formats for systems biology include FASTA for sequence encoding [14], the Systems Biology Markup Language (SBML) for chemical reaction network (CRN) models [15], and Synthetic Biology Open Language (SBOL) for structural and functional information [16]. Model frameworks provide generic functions and objects that can be selectively changed by users to write and simulate custom models within that framework. These include COPASI for stochastic simulations [17], Smoldyn for particle-based reaction-diffusion models [18], COBRA for constraint-based models [19], MCell for Monte Carlo models [20], ECell for stochastic reaction-diffusion models [21], cellPack for spatial packing of molecular shapes [22], CompuCell3D for cellular Potts models [23], PhysiCell for 3D physical models of multicellular systems [24], and BioNetGen for rule-based models [25]. However, standards and frameworks often come with significant constraints, and trying to build a model of novel phenomena almost always comes upon the limitations of what the standards can specify. Committing to one approach can exclude insights that could be gained from others, and to date there is no established method to connect different approaches.

What we therefore need is a software solution for heterogeneous model integration, which allows many modeling efforts to be extended, combined, and simulated together. If standard modeling formats are like an HTML for systems biology, we need an "interface protocol" – analogous to TCP/IP for the Internet – which allows diverse pieces of software to connect, communicate, and synchronize seamlessly into large, complex, and open-ended networks that anyone can contribute to. This software should adopt design principles that free contributors to develop whatever model they want, yet still integrate with and build upon prior work. With this software, collective effort will be more efficiently harnessed to build models with a far wider scope.

This paper introduces Vivarium – software born of the idea that it should be as easy as possible for computational biologists to define any imaginable mechanistic model, combine it with existing models, and execute them together as an integrated multiscale model. Similar approaches have been developed for computer modeling of cyber-physical systems with Ptolemy II [26] and Modelica's Functional Mock-up Interface [27]. Recently, related methods have also begun to be applied to plant modeling with yggdrasil [28], and to synthetic and systems biology with modular environments such as Tellurium [29] and Simbiotics [30].

Our efforts were catalyzed by the needs we had related to whole-cell modeling – flexible integration of diverse frameworks, large simulations, hierarchical embedding, division (to support one model instance becoming two), and parallel execution. We found the solution could be useful more generally, for example with the incorporation of motility and chemotaxis [31], and other functionality we had not even considered before [32]. By explicitly separating the interface that connects models from the frameworks that implement them, Vivarium establishes a modular design methodology that supports flexible model development. The Vivarium interface is applicable to any type of dynamical model – ordinary differential equations (ODEs), stochastic processes, Boolean networks, spatial models, and more – and allows users to plug these models together in integrative, multiscale representations. Here, we describe a multiscale simulation engine, vivarium-core, that combines and runs these models as systems evolving over time. We also present the Vivarium Collective, a GitHub organization of open-source software libraries, with modular models that can be imported into new projects, reconfigured, and recombined to generate entirely new models. The software has been designed to make it straightforward to publish Vivarium models as Python libraries on the Python Package Index (PyPI) to share with the community to plug into existing public or private models.

This paper is organized as follows. Section 2 provides a high-level overview of Vivarium's features and introduces its terminology. Section 3 goes into greater detail as it builds an example system, starting with a deterministic model of unregulated gene expression, and then adding complexity through stochastic multi-time stepping, division, and hierarchical embedding in a shared environment. This example is built up incrementally, highlighting key features of the methodology that enable incremental construction of complex models. Next, Section 4 combines several modeling paradigms into a composite model by interfacing popular modeling frameworks, with a flux-balance model of metabolism simulated with COBRA [19], a stochastic CRN simulated with Bioscrape [33], and a solid-body physics engine for spatial multi-cell interactions simulated with pymunk [34]. All of the examples are available in Jupyter notebooks, designed so that readers can follow along in the code and execute the examples that are described in this paper.

2 Vivarium overview

Vivarium was developed as a synthesis of integrative "whole-cell" modeling and multiscale "agent-based" modeling, implemented in Python with modular software libraries. Vivarium does not include any specific modeling frameworks, but instead focuses on the interface between such frameworks, and provides a powerful multiscale simulation engine that combines and runs them. Users of Vivarium can therefore implement any type of model module they prefer – whether it is a custom module of a specific biophysical system, a configurable model with its own standard format, or a wrapper for an off-the-shelf library. The multiscale engine supports complex agents operating at multiple timescales, and facilitates parallelization across multiple CPUs.

Vivarium's basic elements are *processes* and *stores* (Fig 1a,b; see Table 1 for a list of key definitions), which can be thought of as software implementations of the update functions and state variables of dynamical systems. Consider the difference equation $\Delta x = f(r, x) \cdot \Delta t$. A Vivarium store is a computational object that holds the system's state variables x. A Vivarium process is a computational object that contains the update function f, which describes the inter-dependencies between the variables and how they map from one time (t) to the next ($t + \Delta t$). Processes are configured by parameters r, which give the update functions a distinct shape of mapping from input values to output values

Processes include *ports*, which allow users to wire processes together through variables in shared stores. Variables in a store each have a *schema*, which declare the data type and methods by which updates to the variable are handled (schema types are listed in Supplementary materials, Table 2) – this includes methods such as *updater*, for applying updates to the variables (Supplementary materials, Table 3), and *divider*, for generate daughter states from a mother state (Supplementary materials, Table 4). A *topology* (short for "process-store interaction topology") is a bipartite network that declares the connections between processes and stores, and which is compiled to make a *composite* model with multiple coupled processes. Ideal processes do not have hidden private states, and expose their states by externalizing them in stores. But sometimes private states are unavoidable, and could actually be used to improve performance since

Table 1: Elements of the Vivarium framework. The first use of these terms in the text are shown in *italics*. Some of these elements have software analogs, which are referenced in monospaced `code` format.

Term	Definition
Process	A modular sub-model that encodes a biological mechanism and can be composed with other processes to create a larger composite model. Particular process instances are created as subclasses of the `Process` class.
Store	A collection of state variables read by the processes, which contains methods for applying the processes' updates. State variables in stores that are shared by multiple processes are the only means of communication between those processes. A `Store` class instance is automatically constructed based on the processes' declared variables and their schema.
Port	A named connector on a process that gets connected to a store. Processes can declare one or more port, and the state variables they want to receive through these ports.
Schema	A state variable's declared data type, default value, and methods such as updaters and dividers, by which updates to the variable are handled. Schemas are declared by the processes' and are used to initialize stores at the start of a simulation.
Topology	Short for "process-store interaction topology", this is a bipartite network that declares how to connect processes to stores. It is declared as a Python dictionary for each process, with port names mapped to paths where stores are expected.
Composite	An integrated model with multiple initialized processes, and whose connections to stores are specified by a topology. A `Composite` class has processes and a topology; it is passed to the engine to create the required stores.
Composer	A composer (an instance of the `Composer` class) generates composites by initializing a set of processes and specifying the topology for how they are wired together.
Deriver	`Deriver` is a subclass of `Process`. Deriver instances runs after the other processes and calculate additional state values from other available state variables – for example, concentrations from molecular counts. These are used to offload complexity from the dynamical processes.
Compartment	A store that contains inner stores and processes, rather than the standard store with state variables. Processes can connect to other compartments through boundary stores.
Hierarchy	Short for "compartment hierarchy" – a hierarchical network with nested stores, like a directory structure. A hierarchy's structure can be updated during simulation runtime with update methods such as divide, move, and add.
Engine	vivarium-core's engine is the `Experiment` class. It accepts processes and a topology, creates the stores and connects processes to it, and runs the integrated model forward in time.

they do not have to synchronize. Externalizing state variables in stores allows other processes to wire to the same variables, which couples those processes – they read from and update these same variables as the simulation runs forward in time.

Processes can be connect to stores across a hierarchical representation of nested compartments. Vivarium uses a bigraph formalism [35] – a network with embeddable nodes that can be placed within other nodes, and which can be dynamically restructured. This contrasts with the standard "flat" network that has all nodes at a single level, and usually with fixed connectivity. A *compartment* is a store node with internal nodes, which can include its own internal processes and the standard variable-containing stores (Fig 1d). A *hierarchy* is a place graph, or directory structure, which defines inner/outer nesting relations between compartments (Fig 1e). *Boundary* stores connect processes across compartments in a hierarchy – these make compartments themselves into pluggable models that can be embedded in a hierarchy. Just as with biological systems, compartments are the key to a model's complexity – they organize systems into hierarchies of compartments within compartments, with modules that can be reconfigured and recombined.

The Vivarium *engine* (vivarium-core's `Experiment` class) is provided with the processes and a topology, it constructs the stores based on the processes' declared schemas for each port, assembles the processes and stores into a hierarchy, and executes their interactions in time. Processes can declare their own required time step, and can update their time steps during runtime based on the state of the system. The engine advances the simulation forward by tracking the global time, triggering each process at its respective time step, retrieving updates at the end of each process' time step, and passing these updates to the connected stores (Fig 4c). The structure of a hierarchy is also dynamic and allows for stores, processes, and entire compartments to be created, destroyed, or moved during runtime. This allows for modeling important biological mechanisms that include forming, destroying, merging, division, engulfing, expelling, etc (Fig 4d).

Whereas agents in typical agent-based models follow a minimal set of simple behaviors, Vivarium aims to support large models with thousands of integrated mathematical equations. To accommodate these demands, Vivarium can

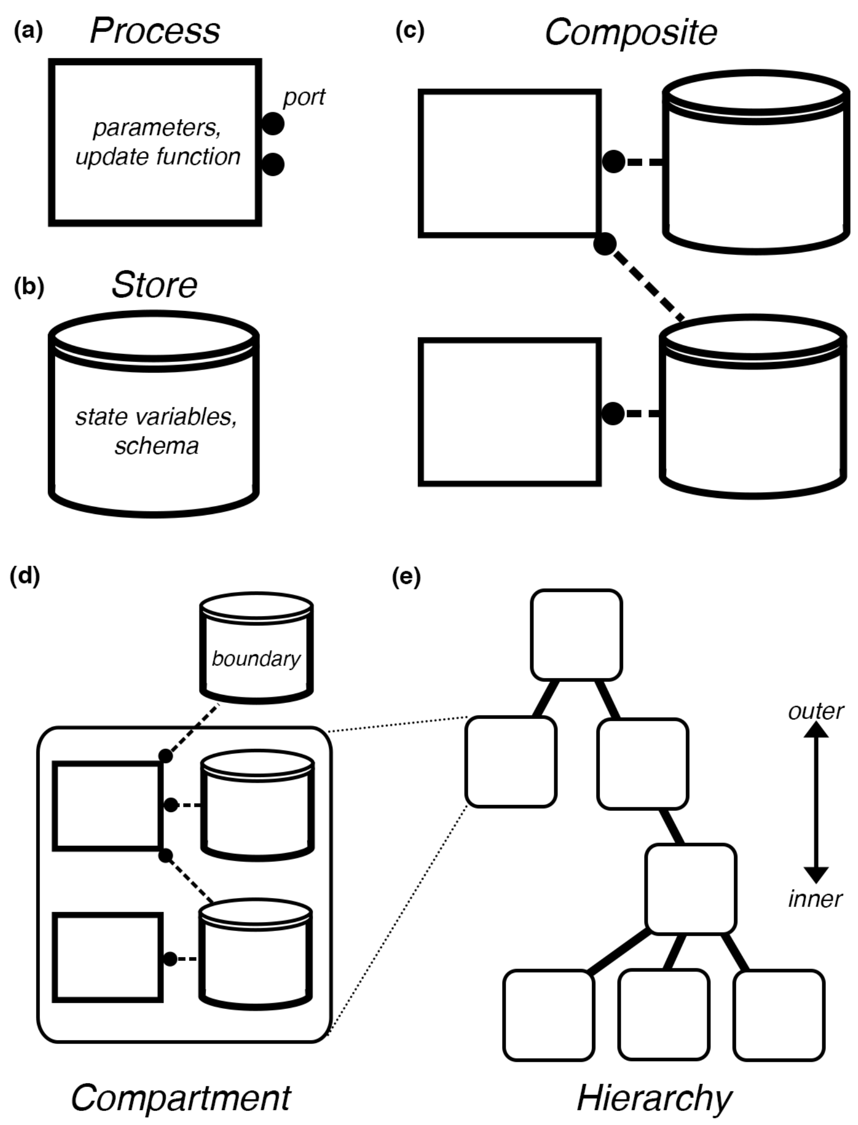


Figure 1: Vivarium's model interface, illustrating the terms in Table 1. (**a**) A *Process*, shown as a rectangular flowchart symbol, is a modular models that contain the parameters, an update function, and ports. (**b**) A *Store*, shown as the flowchart symbol for a database, holds the state variables and *schemas* that determines how to handle updates. (**c**) *Composites* are bundles of processes and stores wired together by a bipartite network called a *topology*, with processes connecting to stores through their ports. (**d**) *Compartments* are processes and stores connected across a single level. Processes can be wired across compartments through *boundary* stores. (**e**) Compartments are embedded in a *hierarchy* – depicted as a hierarchical network with discrete layers. Outer compartments are shown above and inner compartments below.

distribute processes onto different OS processes – which we call *threads* to avoid confusion with Vivarium processes (Fig 4b). Communication between parallel processes on separate threads is mediated by message passing with Python's multiprocessing library. Simulations have run on Google Compute Engine node with hundreds of CPUs [32].

To facilitate collaborative model development, Vivarium provides a modular system that simplifies the incorporation of alternate sub-models. This allows users to 1) write their own processes, composites, and update methods, 2) import libraries with processes developed for different projects, 3) reconfigure and recombine existing processes, and 4) make incremental changes (add, remove, swap, reconfigure) and iterate on model designs that build upon previous work. Auxiliary processes are provided to offload complexity from the main processes. As a Python library, Vivarium is simple to install and import into existing workflows – including Jupyter notebooks, as will be demonstrated in Section 3 and Section 4.

Thus, Vivarium enables us to tackle unprecedented challenges in the modeling of biological systems, bridging methodologies such as whole-cell modeling and agent-based modeling. The plug-in system lowers the barrier to contribution, so that new users can more easily encode new processes and plug them into existing systems. Finally, the vivarium-core code base has been released under the MIT license, permitting open development and re-use. Users can freely build upon existing models by adding new processes, making new predictions, and testing against new data.

3 Interface basics

This section introduces the elements and methodology of Vivarium by working through an example system of unregulated gene expression. We provide supplementary Jupyter notebooks (the links are provided in Supplementary materials section 5) that implement each of the examples in executable code – technical readers are encouraged to have these notebooks open as they read through this section.

The guiding example separates gene expression into two processes: transcription – in which a gene is transcribed to form mRNA, and translation – in which the mRNA is translated to form a protein. We start by representing each of these functions as simply as possible with difference equations, and run them on their own. We then integrate them in a composite model and simulate them together. Next, we replace the deterministic transcription process with a stochastic process, and run a hybrid deterministic/stochastic simulation of gene expression that includes variable time steps. Finally, processes for cell growth and division are added, which allow the system to split into many separate agents that run in parallel in the simulation.

3.1 Transcription process

The transcription process used here is called "Tx", and models mRNA synthesis from DNA. We define a system with a single mRNA species, C, transcribed from a single gene, G. The chemical reaction network (CRN) – which specifies reactants, products, and a set of reactions – takes the form:

$$G \xrightarrow{k_{tx}} G + C,$$

$$C \xrightarrow{k_{deg}} \emptyset.$$

This CRN can be simulated with the difference equation

$$\Delta C = (k_{tx}G - k_{deg}C)\Delta t,$$

with C expressed from G at rate k_{tx}, and degraded at rate k_{deg}. Quantities are in concentrations (mg/mL) – this comes in useful later when converting to concentrations from counts. For pedagogical reasons, this model ignores gene copy number, RNA polymerase abundance, strength of gene promoters, and availability of nucleotides – features that could potentially be added later to improve the model's realism.

Ports. An illustration of Tx is shown in Fig 2a. G and C are read through different ports, "DNA" and "mRNA", which are connected to two different stores, also called "DNA" and "mRNA". By default, a process's ports connect to stores that have the same name – the next subsection demonstrates more complex mappings. For a small model with only two variables splitting the variables into separate ports might seem excessive, but for larger models this is a useful design principle. Generally, port design should be used to organize variables by useful categories: locations such as cytoplasm, membrane, chromosome; molecule groups such as metabolites, proteins, chromosomes; or by other categories such as global variables, fluxes, concentrations.

Process interface. Listing 1 shows Python code for the Tx process – an instance of the `Process` class. Making a process requires implementing the process interface, which involves the following constant and methods: 1) `defaults`: This class constant declares expected parameter names and values – even if only with empty values that get replaced upon initialization. The process constructor (`__init__`) accepts a list of parameters when a process is initialized, which override the defaults. 2) `ports_schema`: This method declares a process' ports ("RNA" and "DNA"), the variables that are accessed through those ports (C and G), and their required schemas (Supplementary materials, Table 2). Any type of value can be used in the schema, such as integers, arrays, or more complex data structures. For novel data types, new schema methods such as updaters and dividers might be required – these are modular and can be defined by users. Updaters and dividers available with vivarium-core are listed in (Supplementary materials, Tables 3 and 4). 3) `next_update()`: This method contains the dynamical model. The steps of this method involves retrieving the variables through the ports, applying the encoded mechanism for the time step's duration, and returning the update for each port.

```
class Tx(Process):
    defaults = {
        'ktx': 1e−2,
        'kdeg': 1e−3}

    def ports_schema(self):
        return {
            'mRNA': {
                'C': {
                    '_default': 100 ∗ units.mg/units.mL,
                    '_updater': 'accumulate',
                    '_emit': True}},
            'DNA': {
                'G': {
                    '_default': 10 ∗ units.mg/units.mL,
                    '_emit': True}}}

    def next_update(self, time step, state):
        # Retrieve the state variables
        G = state['DNA']['G']
        C = state['mRNA']['C']

        # Run the model
        dC = (self.parameters['ktx']∗G −
            self.parameters['kdeg']∗C) ∗ time step

        # Return an update
        return {
            'mRNA': {
                'C': dC}}
```

Listing 1: Python implementation of the minimal transcription process, Tx. This demonstrates the Vivarium process interface. Defaults correspond to default parameter values which can be overwritten in the `Process` constructor. Unit conversions are supported by the pint library [36]. States are Python dictionaries which encode the file structure of a Vivarium model. Updates are similarly returned as in the same hierarchical dictionary format.

Simulating a process. The output of Tx is shown in Fig 2b. Individual processes can be run on their own by the simulation engine, which initializes the stores, runs the simulation, and saves the output. A simple process uses a basic simulation loop, shown in Fig 4a. Each experiment is configured with an emitter, which logs the state of variables marked to emit during runtime – marking a variable to emit can be declared in the process' `port_schema`. If the emitter is connected a database (we use mongoDB), the saved data can be retrieved from the database for visualization and analysis at any time during or after a simulation run.

3.2 Transcription/Translation composite

Next, we integrate the Tx transcription process with a translation process called "Tl", whose implementation is not shown here but is available in the supplementary notebook. Tl takes a similar form to Tx, but with protein X translated from mRNA C and degraded:

$$\Delta X = (k_{tl}C - k_{deg,X}X)\Delta t.$$

As before, units are in concentrations (mg/mL). As a simplifying assumption, translation rate considers ribosome availability, strength of ribosome binding to mRNA, availability of tRNAs and free amino acids to all be part of one lumped constant.

Fig 3a illustrates the composite model called "TxTl", with Tx and Tl both wired to a shared store called mRNA. This couples the two processes, so that mRNAs synthesized by Tx impacts the expression of proteins by Tl.

Composer interface. `Composer` is a class that generates composites. A given composer's inherited `generate()` method calls `generate_processes()` to construct the processes and `generate _topology()` to wire the processes to the stores. Then, it returns a composite that is ready for execution. Making a composer involves the following three class attributes: 1) Composers have their own `defaults` for parameters, which can override the default parameters for individual processes, thus providing easy access for parameter scans and learning algorithms to adjust the full

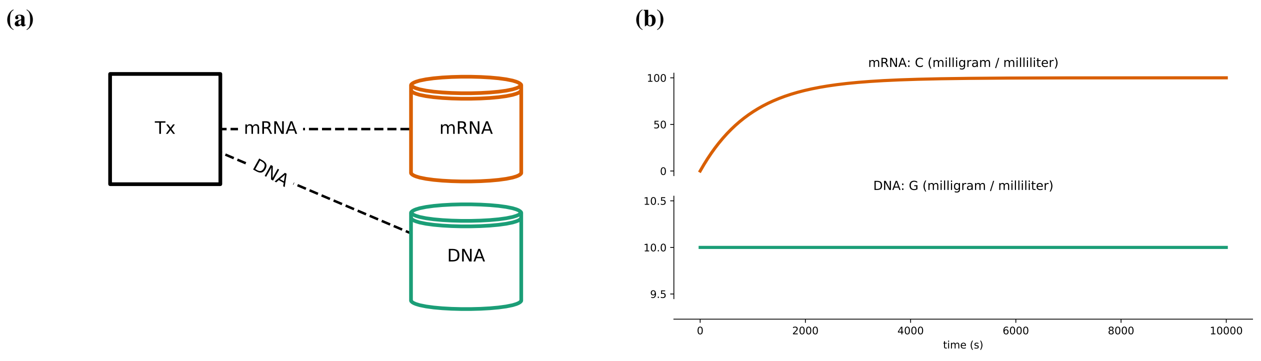


Figure 2: Tx – a transcription process. (**a**) The system's topology, with process Tx wired to two stores – DNA and mRNA – through ports of the same name. (**b**) Simulation output. The DNA G remains fixed at its initial value, and the mRNA C increases up to a steady state.

```
class TxTl(Composer):
    defaults = {
        'Tx': {},
        'Tl': {}}

    def generate_processes(self, config):
        return {
            'Tx': Tx(config['Tx']),
            'Tl': Tl(config['Tl'])}

    def generate_topology(self, config):
        return {
            'Tx': {
                'DNA': 'DNA',
                'mRNA': 'mRNA'},
            'Tl': {
                'mRNA': 'mRNA',
                'Protein': 'Protein'}}
```

Listing 2: Python code for the TxTl Composer. This demonstrates the composer constants and interface methods. Processes are initialized in `generate_processes` and their ports are wired to stores by declaring a topology in `generate_topology`. Listing 5 in the Supplementary materials includes more advanced examples of `generate_topology`, including connecting ports to stores at different levels of the hierarchy, splitting a port into multiple stores, and connecting variables with different names.

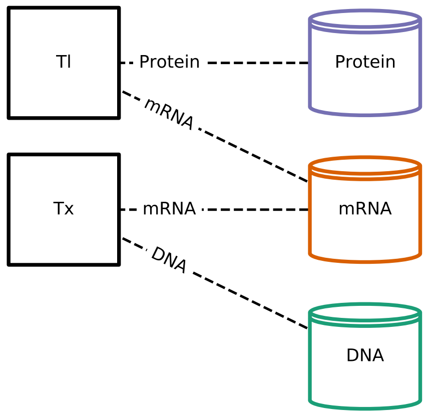
composite's behavior. 2) The `generate_processes()` method constructs a composite's processes in a dictionary, which maps each of their names to the instantiated process objects. 3) The `generate_topology()` method returns a topology dictionary that declares how each process's ports connect to stores in the hierarchy. The TxTl composite is specified in listing 2. Default parameters are empty so the processes will use their own defaults if none are supplied. Each process is constructed in `generate_processes()`, and wired together in `generate_topology()`. Tx's DNA port maps to a store called "DNA", Tl's protein port maps to a store called "Protein", and both Tx and Tl get wired to the same "mRNA" store containing the state variable "C", thus coupling the two processes together.

Simulating a composite. The Vivarium engine runs TxTl to produce the simulation output shown in Fig 3b, where the mRNA reaches a steady-state as before while the protein concentration increases over time following a logistic-like curve.

3.3 Swapping processes for added complexity

Importantly, Vivarium enables users to compare competing models of a given process, simply by exchanging one for the other and simulating the resulting behaviors. The interface introduced above makes it easier to define and integrate

(a)



(b)

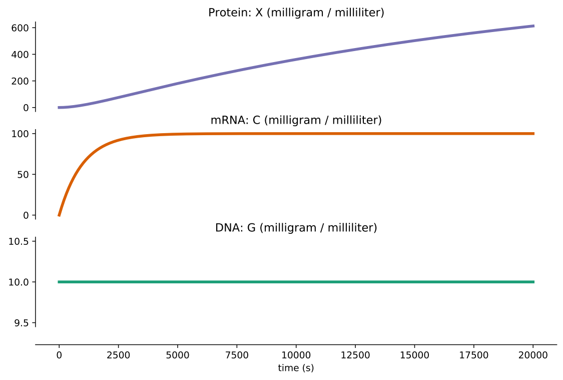


Figure 3: The TxTl composite with Tx transcription and Tl translation. (**a**) The model's topology and process declaration. Tx is the same as in Fig 2, and Tl is a new process with ports for mRNA and protein. Both processes are wired to the same mRNA store, thus coupling them together. (**b**) The simulation output of this model is the same as in Fig 2 showing DNA G and mRNA C, but with added protein X also being expressed.

process modules – now we demonstrate how changes in the sub-models, building off prior model design, can provide additional model functionality.

We begin by replacing the deterministic transcription process, Tx, with a stochastic process called "stochastic Tx". The biological reasoning for this might be that in individual cells many genes are transcribed at low expression rates and synthesize small counts of mRNA, which leads to stochastic behavior. We use the Gillespie algorithm [37] – a discrete and stochastic method for systems with few reactants – to simulate individual reactions. The Gillespie algorithm can be broken into two steps – one for calculating the time which elapses before an event occurs, and the other for determining the nature of that event. Stochastic simulations require variable time steps; for example, the distribution of time steps in a simulation using stochastic Tx is shown in Fig 5c.

The Gillespie algorithm operates on molecular counts – every reaction event increases the counts of the products and decreases the counts of the substrates. Thus, the model needs to convert the molecular counts from the stochastic Tx process to concentrations for input to the Tl process. To perform this conversion, we add an auxiliary *deriver* process. `Deriver` is a subclass of `Process`, whose instances run after the dynamic processes, and derives some state variables from others. The vivarium-core library provides several general-purpose derivers. In the current example we instantiate a deriver called "counts to mg/mL" in Fig 5 – this deriver calculates new concentrations from counts after every step.

Simulating multiple timescales. For processes to operate at different timescales, the simulation engine handles updates on a per-process basis (Fig 4c). At the start of each process' time step, the engine retrieves the process' required time step by calling its `calculate_timestep` method with the current state of the system. By default, the process returns a fixed time step that can be declared in their parameters, but stochastic Tx calculates a new time step by using the Gillespie algorithm. After retrieving the time step, the engine calls the process' `next_update` method with the current state of the system. When the system time reaches the end of the process' time step, it retrieves the update and applies it to the system state. This way, all processes can run at their preferred timescales. With the current version of the engine, the user needs to make sure the time steps of the processes are synchronized with each other to avoid numerical issues. Future versions can introduce a specialized adaptor process to handle the processes' time steps in a way that automatically ensures coordination.

3.4 Hierarchical embedding

Up to this point, each model had a fixed number of processes and stores, with a fixed topology. In contrast, cell division requires a hierarchy with agents embedded in a shared environment, within which the cell agents can grow and divide. The hierarchy needs to launch new processes and stores for each agent created during runtime.

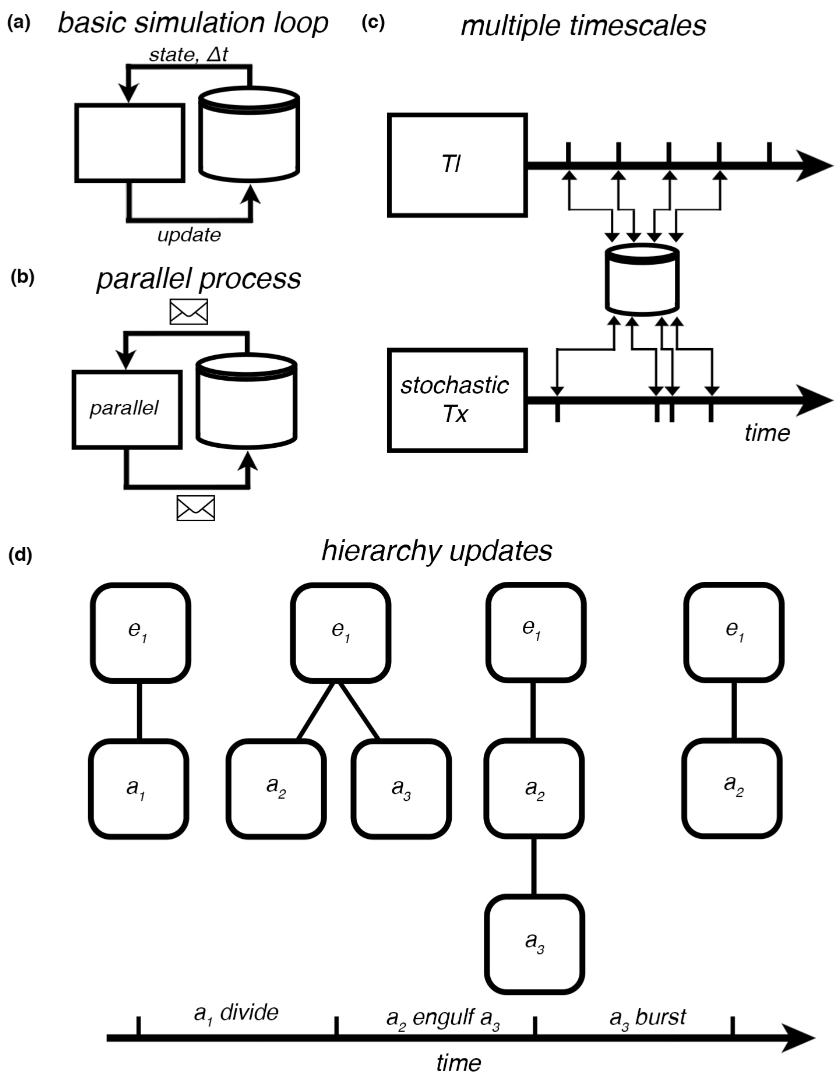


Figure 4: The engine takes the system and runs it forward in time. (**a**) A basic simulation cycle has processes view the states through their ports, run their update function for a given time step, and return an update. (**b**) Any process can run in an OS process, which is placed on a long-running CPU thread, with state views and updates handled by message-passing. This allows for scalable simulation on computers with many CPUs. (**c**) Processes can run at different time scales, and using variable time steps. Here, Tl is shown operating at fixed time steps, and stochastic Tx is operating at variable time steps determined based on the state of the system at each time step's start. (**d**) A series of hierarchy updates depicts a compartment added by a *divide* update, then a compartment subsumed into a neighbor by an *engulf* update, then the engulfed compartment is deleted with a *burst* update. Other hierarchy updates include merge, add, or delete.

Agents and environments. When one compartment is nested in another, the inner compartment can be considered an agent and the outer compartment its environment. Coupling between an agent and an environment is supported by their processes sharing variables in boundary stores. For example, agent processes can update boundary variables required by environmental process such as agent volume, shape, motile forces, and uptake of molecules. Environmental processes can update the boundary conditions of agents' internal processes; for example, local molecular concentrations, and temperature. In the current example, a process called "colony volume" is added to the environment to calculate the volume of all the agents together (Fig 6a). This derived population-level state variable could in principle be used to drive other mechanisms – for example if simulated in a gut microenvironment, bacterial colony volume variable could be used to impact the host's digestion.

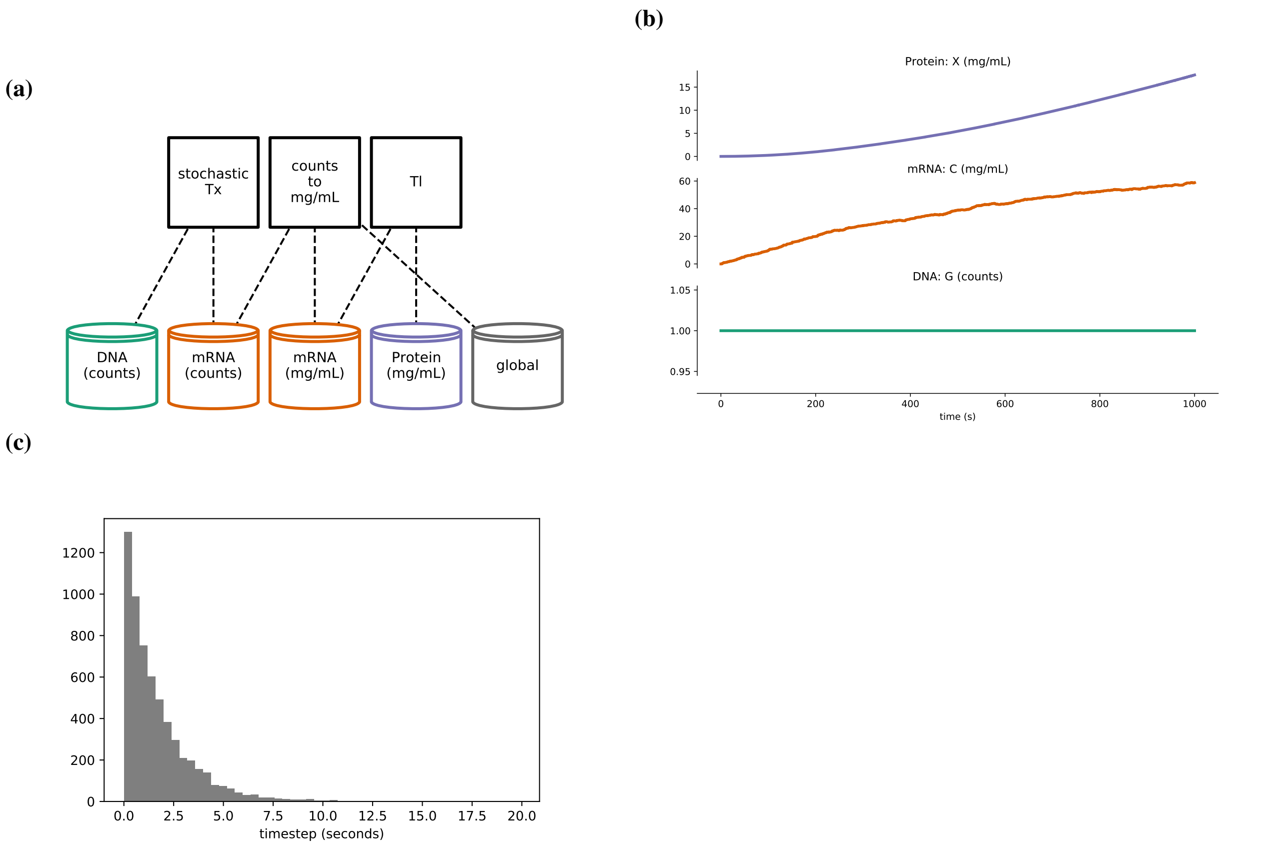


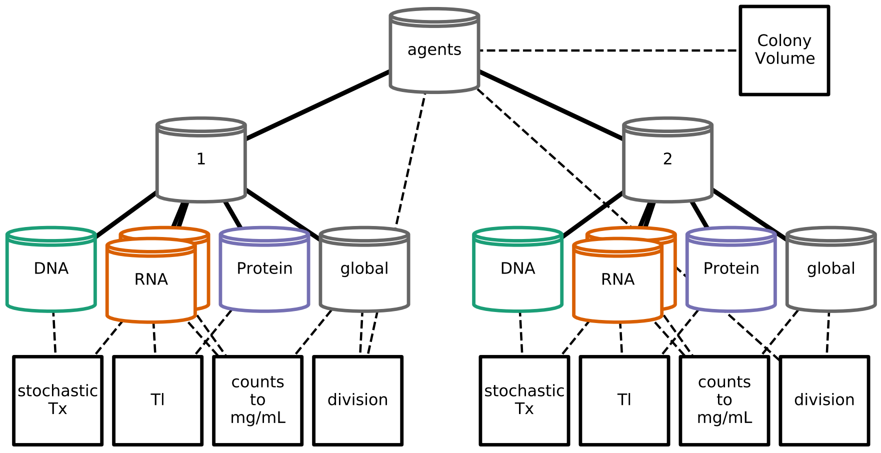
Figure 5: Stochastic transcription with deterministic translation. The stochastic process adjusts its time step based on the total propensity of the system, which is recalculated at every step. (**a**) The topology, showing the stochastic Tx process, the counts to mg/mL concentration deriver, and the deterministic Tl process. Counts to mg/mL is connected to a "global" store, which holds the volume variable required to calculate concentrations. (**b**) The simulation output shows stochastic dynamics of mRNA, and the impact of this stochasticity on the protein concentration. (**c**) Histogram shows the variable time steps resulting from running stochastic Tx on its own for $10,000$ seconds. Tl on its own runs at fixed 1 second intervals.

Advanced process-store topology. Up to this point, the system comprised a single compartment (i.e., a single cell). Accordingly, each topology specified a simple path from each port to a store in the same compartment – a "flat" network. Now, we model the environment as an outer compartment which can contain one or more cells. This requires advanced specifications in a composer's `generate_topology` that connect a processes' ports to stores further up or down the hierarchy. A port connects to a store in the hierarchy by specifying a path, which could go up the hierarchy to stores in outer compartments; or down the hierarchy to stores contained in inner compartments. `generate_topology` also supports splitting ports to draw from variables in separate stores, merging ports to draw from the same store, and aliasing names to variables to connect models with different variable names. Some of these additional techniques are used in Section 4. A few technical examples of these advanced topology methods are included in Supplementary materials section 5.

In the current example, the colony volume process reads the counts of all molecules through an "agents" store, which contains all of the individual agent instances – each with its own DNA, mRNA, and Protein, and global stores. Colony volume is configured to read the volumes in the individual global stores, and calculates total colony volume.

Division. To enable a compartment to divide during runtime, Vivarium provides a division process that is configured with a divide condition, which when true triggers division. A configurable condition means the process could be reused for more sophisticated cell models, for example based on the completion of chromosome segregation or the formation of a septum. Upon division, the mother's variables' states are divided between daughter agents based on those variables' 'divider' schema methods (Table 4).

(a)



(b)

```
# make txtl composite with agent id "0"
agent_id = '0'
txtl_composer = TxTlDivision(agent_config)
agent0 = txtl_composer.generate(
    {'agents': agent_id})

# make a colony composite
colony_composer = ColonyVolume()
colony_composite = colony_composer.generate()

# perform merge
# agent0 in store "0", under "agents"
colony_composite.merge(
    composite=agent0,
    path=('agents', agent_id))
```

(c)

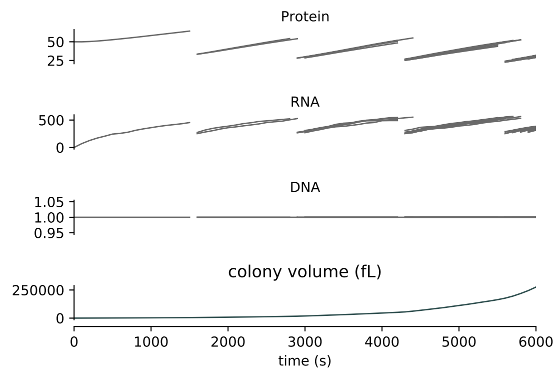


Figure 6: Hierarchical embedding and division. **(a)** Topology plot of the embedded hierarchy, with two cell agents. This plot was generated after one division event occurred, to illustrate how embedding works in Vivarium. Solid edges reflect hierarchy's directory structure, with "agents" at the top level. Dashed edges are topology connections between processes and the stores. **(b)** A script for making a composite model from a TxTl agent and a colony-level process to measure total colony volume. Agent 0 agent is placed within an 'agents' store, while Colony Volume is placed at the top level. The simplicity of the merge operation provides a flexible API for combining different models. **(c)** Simulation output of model over multiple generations of growth and division. Many instances run in parallel by the end of the simulation. The bottom plot shows the total colony volume, calculated from the full set of cell models within the environmental compartment.

For our example system, we initialize agents at 1000 fg, and trigger division when they double that mass. Fig 6c shows how the output over four generations of growth and division, starting off with a single stochastic TxTl instance that splits into two independent instances and then four and then eight – each of which exhibits its own distinctive behavior.

Compartment hierarchy updates. In biological systems, the nesting of compartments can be rearranged over time with behaviors such as engulfing, merging, and division. To support these behaviors, hierarchies in Vivarium can also be restructured during runtime (Fig 4d). There are several built-in hierarchy update methods including move, add, delete, divide, and generate. Processes can trigger these in different combinations to generate a wide range of possible behaviors including merge, two neighboring compartments combine into one; burst, a compartment combines with its environment; engulf, one compartment is moved inside of a neighboring compartment; and expel, a compartment moves to be a neighbor of its outer compartment. All of these are available in vivarium-core.

Fig 6a shows the hierarchy after one division event, with two agents embedded in the top-level "agents" store. This requires the simulation to instantiate new agents – remove the mother agent, and generate two daughter agents with the mother's composer, and an inherited state.

4 Multi-paradigm composites

In the previous section, we were able to introduce many of the core functionalities supported by Vivarium using a simple example. In this section, we demonstrate the power of Vivarium by applying it to much more complex, real-world examples. Specifically, we use Vivarium to integrate several modeling paradigms, building wrapper processes around existing libraries and wiring them together in a large composite simulation. COBRA is used for flux-balance analysis [19], Bioscrape is used to simulate chemical reaction networks [33], and pymunk is used as a solid-body physics engine for spatial multi-cell physics [34]. The result is a multi-paradigm model of an *E. coli* colony with many individual cells in a spatial environment, that collectively undergo a lactose shift in response to glucose depletion. This section describes each process and the integration approach briefly, and then focuses on the final integrated product. For interested readers, we strongly recommend the supplementary Jupyter notebook, which shows the incremental development steps, describes strategies for their integration, and displays the resulting emergent behavior.

When grown in media containing the two sugars glucose and lactose, a colony of *E. coli* will first consume only the glucose until it is depleted; the colony will then enter a lag phase of reduced growth, which is followed by a second phase of growth from lactose uptake. During the glucose growth phase, the expression of the lac operon is inhibited while glucose transporters GalP and PTS are expressed. When external glucose is depleted, cells at first do not have the capacity to import lactose. The lac operon controls three genes: *lacY* (Lactose Permease) which allows lactose to enter the cell, *lacZ* (β-Galactosidase) which degrades the lactose, and *lacA* (Galactoside acetyltransferase) which enables downstream lactose metabolism. Once the operon is activated and proteins are expressed, the metabolism shifts to lactose and growth resumes. See [38] for a more comprehensive overview.

For this example, a flux-balance model of *E. coli* is used to model overall cellular metabolism, while the details of the glucose-lactose regulatory, transport and metabolic circuit are represented by a chemical reaction network model.

4.1 Flux-balance analysis with COBRA

Flux balance analysis (FBA) is an optimization-based metabolic modeling approach that takes network reconstructions of biochemical systems, represented as a matrix of stoichiometric coefficients and a set of flux constraints, and applies linear programming to determine flux distributions, for example those that maximize the production of biomass based on the known composition of metabolic end-products [39]. A strength of FBA is its capacity to simulate whole-network flux distributions using a minimal set of parameters. FBA is made dynamic (called dFBA) by iteratively re-optimizing the objective with updated constraints at every time step [40]; these constraints change with environmental nutrient availability, gene regulation, or enzyme kinetics. Many useful tools related to building and simulating FBA models have been developed and made freely available in the COBRA toolbox, which is also available in python as COBRApy [19].

For this work, we developed a Vivarium process that provides a wrapper around COBRApy, and is located in the vivarium-cobra library (Fig 7a). This process, called "COBRA", can be initialized with a BiGG model from the BiGG model database [41]. BiGG models are genome-scale metabolic models, which are available for dozens of *E. coli* strains, as well as many other cell types. The model used here is *iAF1260b*, which includes 2382 reactions, controlled by 1261 genes, and with an objective that includes the production of 67 molecules.

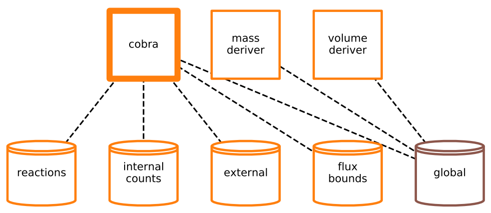
For purposes of integration with Vivarium, we pass the COBRApy results into internal metabolite pools that are available for other processes to utilize. The COBRA process includes a "flux bounds" port, which allows other processes to dynamically modify the flux constraints on the FBA problem. Accordingly, some additional processes were developed so that the COBRA process could support dFBA (shown in Fig 7a). These processes include "local field" to model the external environment with dynamic molecular concentrations, and "mass deriver" to convert the internal metabolite counts into a total mass.

Thus, when COBRA is run as a composite with the local field and mass deriver processes (Fig 7b), it takes up metabolites from the environment and grows its internal pools of metabolites, exponentially increasing in mass and reproducing the expected 40 minute doubling time in minimal glucose media.

4.2 Chemical reaction networks with Bioscrape

To add a CRN network model of transcription, translation, regulation, and the enzymatic activity of the lac operon and its resulting proteins, we turned to a published model [42], with many parameters from [43]. Our model includes all the

(a)



(b)

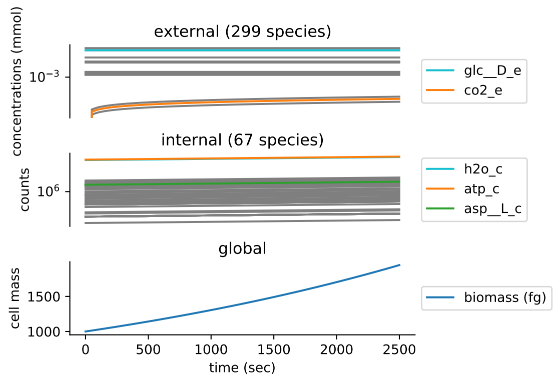


Figure 7: Demonstration of the COBRA composite's dynamic FBA. **(a)** Topology of the COBRA composite, which models metabolism with the COBRA process, and additional processes "local field" and "mass deriver" and help make the system into a dynamic FBA with an external environment, and total compartment mass based on individual metabolite pools and their molecular weights. The COBRA process has ports connected to "reactions", "internal counts", "external" (which are in concentrations), "flux bounds", and "global". **(b)** Simulation of a COBRA composite model, configured with BiGG model *iAF1260b*. Environmental concentrations (top) and internal molecule counts (middle) are plotted in log-scale due to the wide range across molecular species. The internal metabolites are multiplied by their molecular weight and summed to get total biomass (bottom).

same features, except for the addition of different combinatorial conformations of the lacR repressor binding to the lac operon. We converted this model to an SBML format using BioCRNpyler – an open source tool for specifying CRNs [44]. With the model in SBML, we were ran simulations with a Vivarium process built with Bioscrape (Bio-circuit Stochastic Single-cell Reaction Analysis and Parameter Estimation) [33] – a Python package that supports deterministic and stochastic simulations. The "Bioscrape" process is available at the vivarium-bioscrape library. Its topology can be seen in Fig 8a.

Running the lac operon CRN model in isolation shows expected behavior (Fig 8b), with glucose initially being taken up from the environment while lactose is not. Once external glucose is depleted, the lac genes are expressed, concentrations of β-Galactosidase and lactose permease rise, and lactose is brought into the cell and degraded. This is all done smoothly, with continuous dynamics (Fig 8b, left). Using the Bioscrape process also facilitates a stochastic simulation of this CRN (Fig 8b, right) the results of which show lac operon RNA expressed via a randomly-occurring transcription event, followed by expression of the lac proteins, and enabling subsequent import of lactose. For the stochastic model, external nutrients can only exist in a small external environmental volumes – large environments make for large nutrient counts, which slows the stochastic simulator drastically and makes simulations unfeasible. This limitation is corrected by the integrated model, which introduces many separate external locations for nutrients.

4.3 Multicell physics with pymunk and field diffusion

With individual cells being represented by the Bioscrape and COBRA processes, our final step was to model a spatial environment in which these cells can grow, divide, and interact – through physical forces as well as by uptake and secretion of molecules in a shared chemical milieu. The environment is implemented using a composite from the vivarium-multibody library called "lattice", which consists of two processes: "multibody" and "diffusion" (Fig 9a).

The multibody process is a wrapper around the physics engine pymunk [34], which can model individual cell agents as capsule-shaped rigid bodies that can move, grow, and collide. Multibody tracks the following boundary variables for each agent: location, length, width, angle, mass, thrust, and torque. The physics engine applies these variables for the update time step, and returns a new location for each agent. Agents can update volume, mass, and motile forces, thus impacting their movement in the environment. Upon division, the `daughter_location` divider (Table 4) is applied to the location of agents, so that when they divide the daughters are placed end-to-end in the same orientation as the mother.

The diffusion process simulates bounded two-dimensional fields of molecular concentrations. Each lattice site (x, y) holds the local concentrations of any number of molecules, and diffusion simulates how they homogenize across local

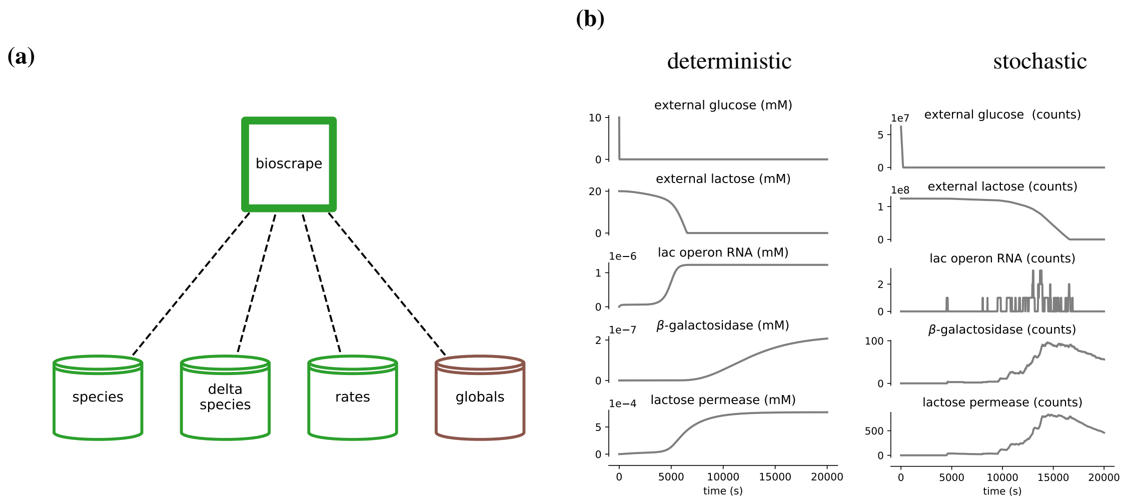


Figure 8: Demonstration of the Bioscrape process on its own. **(a)** Topology of the Bioscrape process, which models a CRN with deterministic (ODE) and stochastic (Gillespie) simulators. Bioscrape has ports connected to "species", "delta species", "rates", and "global". **(b)** Simulation of the Bioscrape process configured with a lac operon model. On the left – a deterministic simulation models the smooth dynamics of molecular concentrations. On the right – A stochastic simulation models the discrete events with molecular counts.

sites. Each agent can uptake and secrete molecules at its position in the field. The implementation uses an adaptor process called "local field", which converts exchanged molecules from the given agent to concentrations at the agent's location.

Fig 9b shows the lattice composite simulated with minimal grow/divide agents. A single initial agent grows and divides to form colonies of many minimal agents in the environment – as they grow, they push against each other via the multibody process, and the colony increases in volume. The agents shown in this minimal simulation do not take up molecules. Therefore, in order to demonstrate the diffusion process, we initialized the system with a concentration gradient, which lessens over time.

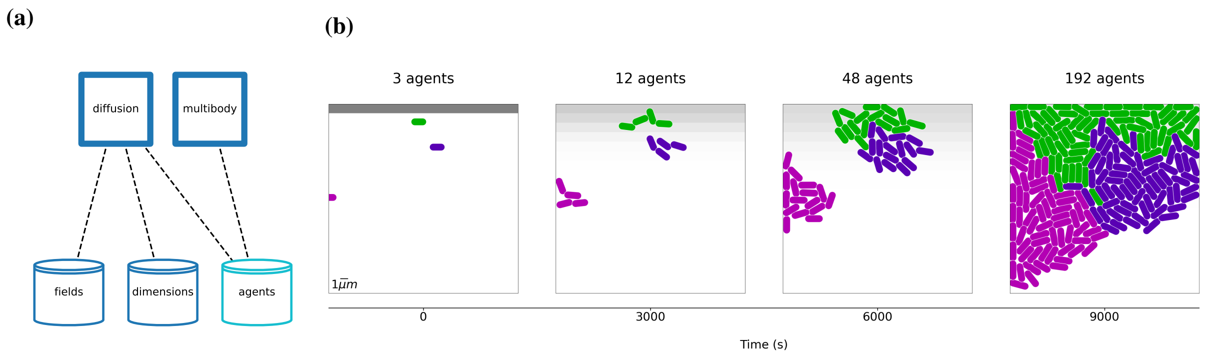


Figure 9: Demonstration of Lattice composite. **(a)** Topology of Lattice composite, with a diffusion process and a multibody process, and ports connected to "fields", "dimensions", and "agents" stores. **(b)** Three grow/divide agents are initialized in the lattice. As the agents grows and divide, the multibody process simulates volume exclusion, which pushes their neighbors away and grows the colony. In this particular case, the agents do not exchange molecules with the external field, but diffusion can be seen by the spread of molecular concentrations initialized at the top row of the field.

4.4 Integration

The initial hierarchy and topology of the model are shown in 10a. Each process described above focuses on a different aspect of cellular physiology and behavior, applies a different mathematical representation, and formats its data by

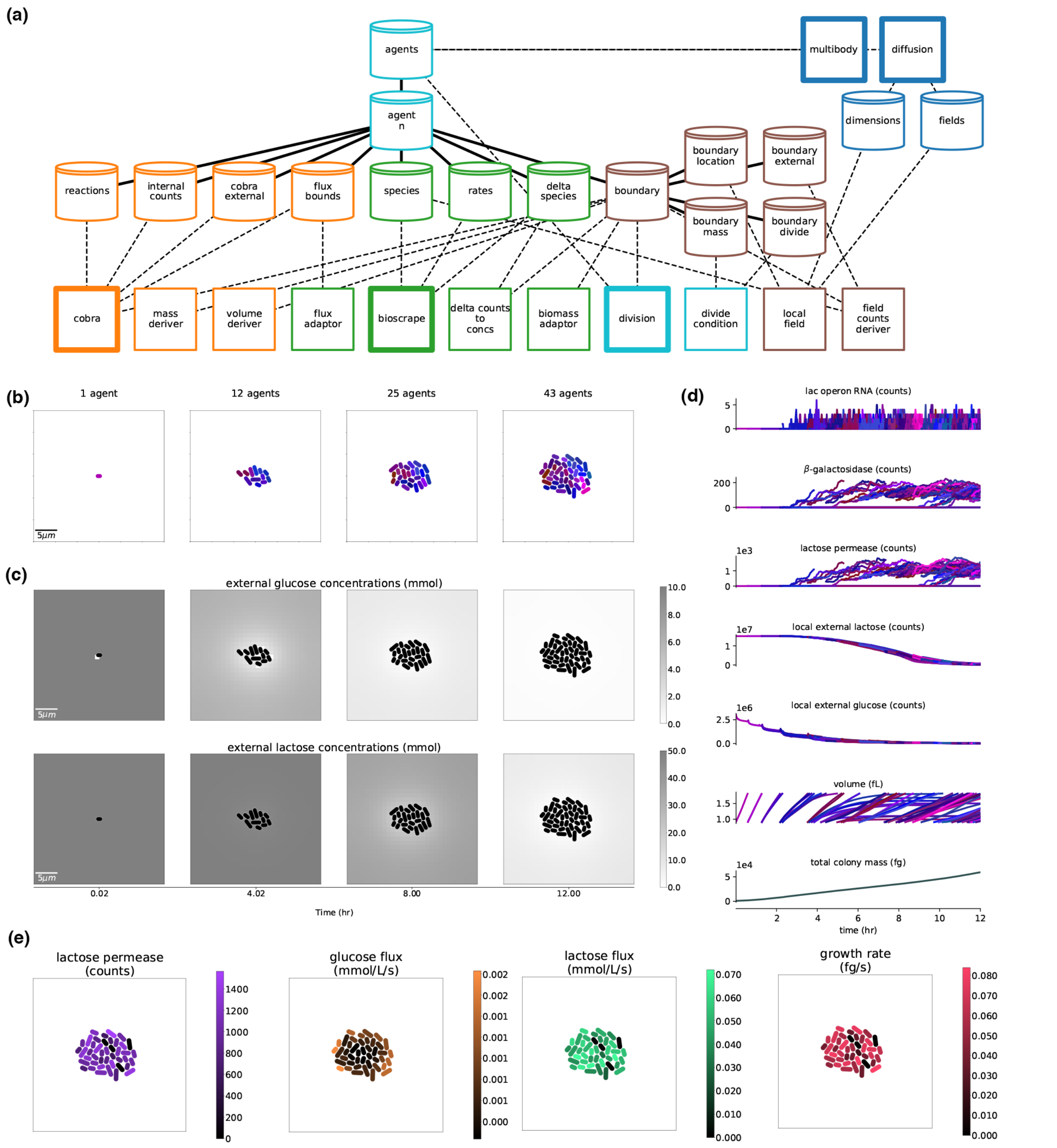


Figure 10: The full, integrated, stochastic version of the model in the Lattice environment. **(a)** Hierarchy and topology of the full composite model. The processes and stores are colored by paradigm, with orange nodes associated with COBRA, green nodes associated with Bioscrape, light blue with division, dark blue with the environment, and brown nodes associated with the boundary between cell and environment. Main processes are highlighted with a thicker outline. **(b)** Snapshots of the colony throughout a simulation. Cells colored according to phylogeny with similar colors indicating more closely related cells. **(c)** The external lactose and glucose concentration fields over the course of the simulation. **(d)** Multi-generational timeseries, with variables from all cells shown over time. These colors correspond to the phylogeny colors of (b). **(e)** Snapshots of the final simulation state ($12.0hr$) , with various cell states tagged.

different standards. Integrating them requires data conversions, and complex mapping of assumptions about their shared variables. This is achieved with a set of *adaptor* processes which convert between the expected units, reference frames, name spaces, data formats, and other representations. Adaptor processes required for integrating the Bioscrape and COBRA processes with the lattice composite include processes "local field", "mass deriver", "volume deriver", "flux adaptor", "biomass adaptor".

The COBRA process' metabolism and Bioscrape process' gene expression and transport are coupled through a "flux bounds" store (Fig 10a), with the Bioscrape process setting uptake rates with the flux adaptor process, and the COBRA process using them as flux constraints on the FBA problem. The Bioscrape process calculates deltas for each reaction's substrates and products with its stochastic kinetic simulator. These deltas are then used by the flux adaptor to calculate a time-averaged flux, which it passes to the flux bounds store. The COBRA process uses these values to constrain the FBA problem's fluxes bounds and thus shapes the resulting flux distribution and overall growth. The Bioscrape and COBRA processes are also coupled through the transporter proteins and internal metabolite pools that are built up by metabolism. Changes in the internal metabolite pools trigger the expression of genes (*lac* genes in this example), which in turn influence the kinetic transport rates. Thus, a causal loop is implemented between the Bioscrape and COBRA processes.

The stochastic versus deterministic versions of the composite require different adaptor processes for converting between counts and concentrations – for example, a "field counts deriver" process was required for the stochastic Bioscrape process to read external counts of glucose and lactose in a cell's local environment based on the lattice resolution. The deterministic model runs much faster and for a larger local environment volume – as the local environment volume increases, this translates to many counts of glucose and lactose that the stochastic model has to simulate with the Gillespie algorithm, which is a time-intensive method. Thus, on its own, the stochastic model is limited to very small external environments and rapid depletion of nutrient. This challenge was overcome by partitioning the environment with the spatial processes described below. The deterministic simulation has the CRN dilution rate parameter coupled to the COBRA-determined growth rate with a process called "dilution rate adaptor" – this is not needed for stochastic simulations, since dilution is handled upon division with the binomial division of molecular counts. Future work would benefit from having two CRN process running in parallel – one with a stochastic simulators for reactions with small molecular counts, and one with a deterministic simulator for large counts.

To model cell division, "division" and "divide condition" processes were added to read the agent's individual cell mass, and trigger division when it reaches $2000fg$. The mass deriver process takes the counts and molecular weights of these molecules (produced by the COBRA process), and calculates the total cell mass. The volume deriver process then calculates various cell shape properties of the cell from its mass, including volume, length, and width. These will be used by the spatial environment. The divide condition process connects to the mass variable directly, and waits for it to cross a configured threshold value. When this threshold is passed, a $division = True$ flag is set, which division sees and performs the hierarchy update that terminates the mother agent and generates two daughter agents – calling each variable's "divider" variable to do so.

The environment consists of 2D arrays of concentration values for glucose and lactose, which set the local external environments for individual cell agents. The local field process converts COBRA process-generated molecular exchanges into concentration changes in spatial fields. The diffusion process takes the resulting fields, diffuses them, and updates the local external variables for each agent, so that the agent's only experience the concentrations at their given location. The agents grow, when they hit the division threshold they divide, their daughter cells are placed end-to-end, and their growth pushes upon neighbors with the multibody process. Thus, cell growth and division lead to the emergence of a colony with many individuals.

The full model (using the stochastic version of the glucose-lactose circuit), was used to simulate a glucose-lactose diauxic shift (Fig 10b-e). This simulation is configured with very little initial glucose, so that the onset of lactose metabolism can be triggered within a few hours of simulation time. As the initial cell grows from a single agent through multiple generations (Fig 10b), it initially takes up glucose and not lactose (Fig 10c,d). The glucose field sites occupied by the cells are depleted by cellular uptake, but replenished by diffusion from neighboring lattice sites (Fig 10c, top). The spatial variation in the environment leads to cells experiencing different local concentrations of glucose and lactose. In response some lac operon RNA is expressed, both stochastically and in response to the local environment in certain cells (Fig 10d). Stochastic expression of the lac genes allows lactose to be taken into the cells (Fig 10c, bottom), but with different levels of the lac proteins, leading to heterogeneous uptake rates of glucose and lactose, as well as the cellular growth rate (Fig 10e). By the end of the simulation, there is still slight glucose uptake; the colony is growing very slowly and is still in lag phase of growth, but lactose-driven growth has become the main driver of colony growth.

5 Discussion

This paper introduces Vivarium – a software tool for integrative multiscale modeling designed to make it as easy as possible for users to access, modify, build upon, and integrate existing models. We demonstrated the integration of several diverse frameworks including deterministic and stochastic models, constraint-based models, hierarchical embedding, division, solid-body physics, and spatial diffusion. Each process was developed and tested independently, and was then wired into a larger composite model that coupled the diverse mechanistic representations. We also introduced the Vivarium Collective, a set of software libraries which include the multiscale engine, vivarium-core, as well as separate process libraries vivarium-cobra, vivarium-bioscrape, and vivarium-multibody. When released as a Python package, libraries can be imported into other projects, re-configured, combined with other processes, and simulated in large experiments. This functional separation helps breaks up the model development methodology and opens module development to a larger community of contributors.

Vivarium is general. This was demonstrated by the large number of different models representations applied at different levels of granularity, including processes that interface existing off-the-shelf modeling libraries such as COBRA, Bioscrape, and pymunk. Scientists and developers can continue working on their preferred modeling framework, yet plug into broader integrative models with many potential contributors with each lending their own expertise. Its broad applicability has the potential to unify the many diverse research efforts, and bring about a new integrative modeling paradigm composed of many constituent modeling frameworks.

Vivarium is flexible. Its emphasis on the interface between models simplifies the incorporation of alternate sub-models, which supports incremental, modular development. The keys to this feature are the smaller adaptor processes, which help convert the units, reference frames, name spaces, data formats, and other representations that are expected by the main mechanistic processes. We demonstrated Vivarium's flexibility by substituting a stochastic model for a deterministic one in Sections 3 and 4, but Vivarium is capable of substituting a wide range of other modeling types. By open-sourcing this project, we hope scientists will build upon existing models by adding new processes, making new predictions, and testing against new data.

Finally, Vivarium is scalable. An integrated model can be built with arbitrary embedded hierarchies of compartments that could represent mechanisms from the molecular to the ecological, and which can be simulated at multiple time steps. Its processes can be distributed across a computer architecture with many CPUs, and so support many parallel mechanisms running concurrently at their preferred timescales. At the time of this paper's writing, modeling even the most minimal cells remains a large-scale effort requiring many scientific contributors. Vivarium was originally conceived and developed specifically to address the challenges associated with whole-cell modeling; namely, integrating a large number of disparate models into one unified whole. However, it quickly became clear that Vivarium would also facilitate even more ambitious goals, such as building the first "whole-colony" computational models that can mechanistically link expression of individual proteins to a population-level phenotype [32].

As multi-scale models such as this one are further developed and expanded, we hope that Vivarium and its successors will be used to model cell populations, tissues, organs, or even entire organisms and their environments – all of which are based on a foundation of molecular and cellular interactions, represented using the most appropriate mathematics, and integrated together in unified composite systems.

Acknowledgements

We thank Richard Murray for discussion and feedback on the paper, and the Build-A-Cell community for organizing fascinating workshop discussions that helped motivate the use of Vivarium as a general modeling tool.

Funding

This work was supported by the Paul G. Allen Frontiers Group via an Allen Discovery Center at Stanford, as well as NIGMS of the National Institutes of Health under award number F32GM137464 to E.A., and NSF grant CBET-1903477 to W.P. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

[1] Amos Tanay and Aviv Regev. Scaling single-cell genomics from phenomenology to mechanism. *Nature*, 541(7637):331–338, 2017.

[2] Robert D Phair. Mechanistic modeling confronts the complexity of molecular cell biology. *Molecular biology of the cell*, 25(22):3494–3496, 2014.

[3] Jason H Yang, Sarah N Wright, Meagan Hamblin, Douglas McCloskey, Miguel A Alcantar, Lars Schrübbers, Allison J Lopatkin, Sangeeta Satish, Amir Nili, Bernhard O Palsson, et al. A white-box machine learning approach for revealing antibiotic mechanisms of action. *Cell*, 177(6):1649–1661, 2019.

[4] Barak Raveh, Liping Sun, Kate L White, Tanmoy Sanyal, Jeremy Tempkin, Dongqing Zheng, Kala Bharath Pilla, Jitin Singla, ChenXi Wang, Jihui Zha, et al. Bayesian metamodeling of complex biological systems across varying representations. *bioRxiv*, 2021.

[5] Isseki Yu, Takaharu Mori, Tadashi Ando, Ryuhei Harada, Jaewoon Jung, Yuji Sugita, and Michael Feig. Biomolecular interactions modulate macromolecular structure and dynamics in atomistic model of a bacterial cytoplasm. *Elife*, 5:e19274, 2016.

[6] Adam Arkin, John Ross, and Harley H McAdams. Stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected Escherichia coli cells. *Genetics*, 149(4):1633–1648, 1998.

[7] Jeremy S Edwards, Rafael U Ibarra, and Bernhard O Palsson. In silico predictions of Escherichia coli metabolic capabilities are consistent with experimental data. *Nature biotechnology*, 19(2):125–130, 2001.

[8] Pontus Melke, Patrik Sahlin, Andre Levchenko, and Henrik Jönsson. A cell-based model for quorum sensing in heterogeneous bacterial colonies. *PLoS Comput Biol*, 6(6):e1000819, 2010.

[9] Jonathan R Karr, Jayodita C Sanghvi, Derek N Macklin, Miriam V Gutschow, Jared M Jacobs, Benjamin Bolival Jr, Nacyra Assad-Garcia, John I Glass, and Markus W Covert. A whole-cell computational model predicts phenotype from genotype. *Cell*, 150(2):389–401, 2012.

[10] D N Macklin, T A Ahn-Horst, H Choi, N A Ruggero, J Carrera, J C Mason, G Sun, E Agmon, M M DeFelice, I Maayan, K Lane, R K Spangler, T E Gillies, M L Paull, S Akhter, S R Bray, D S Weaver, I M Keseler, P D Karp, J H Morrison, , and M W Covert. Simultaneous cross-evaluation of heterogeneous E. coli datasets via mechanistic simulation. *Science*, 369(6502):eaav3751, 2020.

[11] Andrew B Ward, Andrej Sali, and Ian A Wilson. Integrative structural biology. *Science*, 339(6122):913–915, 2013.

[12] William R Harcombe, William J Riehl, Ilija Dukovski, Brian R Granger, Alex Betts, Alex H Lang, Gracia Bonilla, Amrita Kar, Nicholas Leiby, Pankaj Mehta, et al. Metabolic resource allocation in individual microbes determines ecosystem interactions and spatial dynamics. *Cell reports*, 7(4):1104–1115, 2014.

[13] Ines Thiele, Swagatika Sahoo, Almut Heinken, Johannes Hertel, Laurent Heirendt, Maike K Aurich, and Ronan MT Fleming. Personalized whole-body models integrate metabolism, physiology, and the gut microbiome. *Molecular systems biology*, 16(5):e8982, 2020.

[14] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[15] Sarah M Keating, Dagmar Waltemath, Matthias König, Fengkai Zhang, Andreas Dräger, Claudine Chaouiya, Frank T Bergmann, Andrew Finney, Colin S Gillespie, Tomáš Helikar, et al. SBML level 3: an extensible format for the exchange and reuse of biological models. *Molecular Systems Biology*, 16(8), 2020.

[16] Bryan Bartley, Jacob Beal, Kevin Clancy, Goksel Misirli, Nicholas Roehner, Ernst Oberortner, Matthew Pocock, Michael Bissell, Curtis Madsen, Tramy Nguyen, et al. Synthetic biology open language (SBOL) version 2.0. 0. *Journal of integrative bioinformatics*, 12(2):902–991, 2015.

[17] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. COPASI—a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[18] Steven S Andrews, Nathan J Addy, Roger Brent, and Adam P Arkin. Detailed simulations of cell biology with Smoldyn 2.1. *PLoS Comput Biol*, 6(3):e1000705, 2010.

[19] Ali Ebrahim, Joshua A Lerman, Bernhard O Palsson, and Daniel R Hyduke. COBRApy: constraints-based reconstruction and analysis for python. *BMC systems biology*, 7(1):74, 2013.

[20] Joel R Stiles, Thomas M Bartol, et al. Monte Carlo methods for simulating realistic synaptic microphysiology using MCell. *Computational neuroscience: realistic modeling for experimentalists*, pages 87–127, 2001.

[21] Satya Arjunan and Masaru Tomita. Modeling reaction-diffusion of molecules on surface and in volume spaces with the E-Cell system. *Nature Precedings*, pages 1–1, 2009.

[22] Graham T Johnson, Ludovic Autin, Mostafa Al-Alusi, David S Goodsell, Michel F Sanner, and Arthur J Olson. cellPACK: a virtual mesoscope to model and visualize structural systems biology. *Nature methods*, 12(1):85–91, 2015.

[23] Maciej H Swat, Gilberto L Thomas, Julio M Belmonte, Abbas Shirinifard, Dimitrij Hmeljak, and James A Glazier. Multi-scale modeling of tissues using CompuCell3D. *Methods in cell biology*, 110:325–366, 2012.

[24] Ahmadreza Ghaffarizadeh, Randy Heiland, Samuel H Friedman, Shannon M Mumenthaler, and Paul Macklin. PhysiCell: an open source physics-based cell simulator for 3-d multicellular systems. *PLoS computational biology*, 14(2):e1005991, 2018.

[25] James R Faeder, Michael L Blinov, and William S Hlavacek. Rule-based modeling of biochemical systems with BioNetGen. In *Systems biology*, pages 113–167. Springer, 2009.

[26] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[27] Torsten Blockwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings*, 2012.

[28] Meagan Lang. yggdrasil: a python package for integrating computational models across languages and scales. *in silico Plants*, 1(1):diz001, 2019.

[29] Kiri Choi, J Kyle Medley, Matthias König, Kaylene Stocking, Lucian Smith, Stanley Gu, and Herbert M Sauro. Tellurium: an extensible python-based modeling environment for systems and synthetic biology. *Biosystems*, 171:74–79, 2018.

[30] Jonathan Naylor, Harold Fellermann, Yuchun Ding, Waleed K Mohammed, Nicholas S Jakubovics, Joy Mukherjee, Catherine A Biggs, Phillip C Wright, and Natalio Krasnogor. Simbiotics: a multiscale integrative platform for 3d modeling of bacterial populations. *ACS Synthetic Biology*, 6(7):1194–1210, 2017.

[31] Eran Agmon and Ryan K Spangler. A multi-scale approach to modeling E. coli chemotaxis. *Entropy*, 22(10):1101, 2020.

[32] Christopher J Skalnik, Eran Agmon, Ryan K Spangler, Lee Talman, Jerry H Morrison, Shayn M Peirce, and Markus W Covert. Whole-colony modeling of Escherichia coli. *bioRxiv*, 2021.

[33] Anandh Swaminathan, William Poole, Victoria Hsiao, and Richard M Murray. Fast and flexible simulation and parameter estimation for synthetic biology using bioscrape. 2017.

[34] Victor Blomqvist. Pymunk. http://www.pymunk.org/, 2007–2019.

[35] Robin Milner. *The space and motion of communicating agents*. Cambridge University Press, 2009.

[36] pint. pint. https://pint.readthedocs.io/en/stable/, 2021.

[37] Daniel T Gillespie. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55, 2007.

[38] Moisés Santillán and Michael C Mackey. Quantitative approaches to the study of bistability in the lac operon of Escherichia coli. *Journal of The Royal Society Interface*, 5(suppl_1):S29–S39, 2008.

[39] Jeffrey D Orth, Ines Thiele, and Bernhard Ø Palsson. What is flux balance analysis? *Nature biotechnology*, 28(3):245–248, 2010.

[40] Amit Varma and Bernhard O Palsson. Stoichiometric flux balance models quantitatively predict growth and metabolic by-product secretion in wild-type Escherichia coli W3110. *Applied and environmental microbiology*, 60(10):3724–3731, 1994.

[41] Zachary A King, Justin Lu, Andreas Dräger, Philip Miller, Stephen Federowicz, Joshua A Lerman, Ali Ebrahim, Bernhard O Palsson, and Nathan E Lewis. BiGG models: A platform for integrating, standardizing and sharing genome-scale models. *Nucleic acids research*, 44(D1):D515–D522, 2016.

[42] M Santillán, MC Mackey, and ES Zeron. Origin of bistability in the lac operon. *Biophysical journal*, 92(11):3830–3842, 2007.

[43] Patrick Wong, Stephanie Gladney, and Jay D Keasling. Mathematical model of the lac operon: inducer exclusion, catabolite repression, and diauxic growth on glucose and lactose. *Biotechnology progress*, 13(2):132–143, 1997.

[44] William Poole, Ayush Pandey, Zoltan Tuza, Andrey Shur, and Richard M Murray. BioCRNpyler: Compiling chemical reaction networks from biomolecular parts in diverse contexts. *BioRxiv*, 2020.

Supplementary materials

Software availability

Vivarium-core, the interface classes and multiscale simulation engine:
`https://github.com/vivarium-collective/vivarium-core`

The Vivarium Collective Github organization is an online location for various vivarium libraries, including all libraries used for this paper.
`https://github.com/vivarium-collective`

Vivarium-notebooks is the repository developed for this paper, with Jupyter notebooks and Python files.
`https://github.com/vivarium-collective/vivarium-notebooks`

Vivarium interface basics notebook.
`https://github.com/vivarium-collective/vivarium-notebooks/blob/main/notebooks/Vivarium_interface_basics.ipynb`

Multi-paradigm composites notebook.
`https://github.com/vivarium-collective/vivarium-notebooks/blob/main/notebooks/Multi-Paradigm-Composites.ipynb`

Notebook for constructing the deterministic and stochastic lac operon CRN models with BioCRNpyler.
`https://github.com/vivarium-collective/vivarium-notebooks/blob/main/notebooks/Lac_Operon_CRN.ipynb`

Tables

This section includes several tables. Table 2 covers the different types of schemas introduced in Section 3. Updater and divider schema have their own tables (3 and 4) with methods provided by vivarium-core. In addition to these provided methods, custom updaters and dividers can easily be defined by the user for any type of desired state. Table 5 includes the vivarium libraries used for this paper – all of which are freely available at the Vivarium Collective, and on PyPI.

Table 2: Schema. A process' `ports_schema` method declares the schemas for variables connected to each port. These are used to construct the stores, which apply the declared methods during runtime.

Attribute	Method
_default	The default value of the state variable if no initial value is provided. This also sets the data type of the variable, including units.
_updater	How to apply state variable updates. Available updaters are listed in Table 3.
_divider	How to divide the state variable's values between daughter cells. Available dividers are listed in Table 4.
_emit	A Boolean value that sets whether to log this variable to the simulation database for later analysis.
_properties	User-defined properties such as molecular weight. These can be used for calculating variables such as total system mass.

Table 3: Updaters available in vivarium-core. Updaters are schema methods by which an update from a process is applied to a variable's value.

Name	Function
accumulate	The default updater. Add the update value to the current value.
set	The update value becomes the new current value.
merge	Update an existing dictionary with new values, and add any newly declared keys.
null	Do not apply the update.
nonnegative_accumulate	Add the update value to the current value, and set to 0 if the result is negative.

Table 4: Dividers available in vivarium-core. Dividers are schema methods by which a variable's value is divided when division is triggered.

Name	Function
set	The default divider. Daughters get the same value as the mother.
binomial	Sample the first daughter's value from a binomial distribution of the mother's value, and the second daughter gets the remainder.
split	Divide the mother's value in two. Odd integers will make one daughter receive 1 more than the other daughter.
split_dict	Splits a dictionary of $key : value$ pairs, with each daughter receiving a dictionary with the same keys, but with each value split.
zero	Daughter values are both set to 0.
no_divide	Asserts that this value should not be divided.

Table 5: Vivarium libraries used in this project.

Library	Description
vivarium-bioscrape	For chemical reaction networks with SBML.
vivarium-cobra	For constraint-based models of metabolism, with BiGG model interface.
vivarium-multibody	Lattice composite model used for spatial multi-cell interactions.
vivarium-template	A template library that can be cloned to get a new vivarium project started.
vivarium-scripts	A collection of scripts to access and modify saved simulations from the database.

Code examples

This supplementary subsection includes some code listings demonstrating more advanced methods for specifying models with Vivarium.

Listing 3: Declaring custom updaters and dividers in `port_schema`. Updaters are functions that take a `current_value` and a `update_value`, and return the new value. Dividers are functions that take a `mother_value`, and a `state`, and return two values in a list – one for each daughter.

```
# updater that returns a random value
def random_updater(current_value, update_value):
    return random.random()

# divider that returns a random value for each daughter
def random_divider(mother_value, state):
    return [
        random.random(),
        random.random()]

def port_schema(self):
    ports = {
        'port1': {
            'variable1': {
                '_default': 1.0
                '_updater': {
                    'updater': random_updater
                    }
                '_divider': {
                    'divider': random_divider
            }
        }
    }
    return ports
```

Listing 4: Using glob ('*') schema to declare expected sub-store structure. In this example, `port1` is connected to sub-stores specified by a glob schema. This allows the process to read anything that port1 connects to which adheres to its declared schema. Sub-stores can be added and removed during runtime, and the process will see it.

```
def port_schema(self):
    ports = {
        'port1': {
            '*': {
                '_default': 1.0
            }
        }
    }
    return ports
```

Listing 5: Declaring process-store connections with `generate_topology`. A topology is a Python dictionary with keys for processes, and subkeys for their ports which map to paths at which they will connect to stores. A flat network does not require a path, just a store name at the same level. There are three different syntaxes possible for declaring paths – Multics string, Unix string, and Unix tuple. The default used within the engine is a Unix tuple, but the other syntaxes can be used in a composer to simplify code readability, and are converted to Unix tuples.

```
def generate_topology(self, config):
    topology = {
        # Multics string example
        'process1': {
            'port1': 'path>to>store', # connect port1 to inner compartment
            'port2': '<outer_store' # connect port2 to outer compartment
        },
        # Unix string example
        'process2': {
            'port1': 'path/to/store', # connect port1 to inner compartment
            'port2': '../outer_store' # connect port2 to outer compartment
        },
        # Unix tuple example
        'process3': {
            'port1': ('path','to','store'), # connect port1 to inner compartment
            'port2': ('..','outer_store') # connect port2 to outer compartment
        }
    }
    return topology
```

Listing 6: Splitting a port into multiple stores with `generate_topology`. Variables read through the same port can come from different stores. To do this, the port is mapped to a dictionary with a `_path` key that specifies the path to the default store. Variables that need to be read from different stores each get their own path in that same dictionary. This same approach can be used to remap variable names, so different processes can use the same variable but see it with different names.

```
def generate_topology(self, config):
    topology = {
        # split a port into multiple stores
        'process1': {
            'port': {
                '_path': ('path_to','default_store'),
                'rewired_variable': ('path_to','alternate_store')
            }
        }
        # mapping variable names in process to different name in store
        'process2': {
            'port': {
                '_path': ('path_to','default_store'),
                'variable_name': 'new_variable_name'
            }
        }
    }
    return topology
```