*Bioinformatics*, 31(20), 2015, 3315–3321
doi: 10.1093/bioinformatics/btv363
Advance Access Publication Date: 17 June 2015
Original Paper

OXFORD

Systems biology

# libRoadRunner: a high performance SBML simulation and analysis library

**Endre T. Somogyi[1,*], Jean-Marie Bouteiller[2], James A. Glazier[1], Matthias König[3], J. Kyle Medley[4], Maciej H. Swat[1] and Herbert M. Sauro[4,*]**

[1]Biocomplexity Institute and Department of Physics, Indiana University, Bloomington, IN, 47405, USA, [2]Biomedical Engineering Department, University of Southern California, Los Angeles, CA, 90089, USA, [3]Department of Computational Systems Biochemistry, University Medicine Charité Berlin, 10117, Berlin, Germany and [4]Department of Bioengineering, University of Washington, Seattle, WA 98195, USA

*To whom correspondence should be addressed.
Associate Editor: Jonathan Wren

Received on February 24, 2015; revised on May 20, 2015; accepted on June 5, 2015

## Abstract

**Motivation:** This article presents libRoadRunner, an extensible, high-performance, cross-platform, open-source software library for the simulation and analysis of models expressed using Systems Biology Markup Language (SBML). SBML is the most widely used standard for representing dynamic networks, especially biochemical networks. libRoadRunner is fast enough to support large-scale problems such as tissue models, studies that require large numbers of repeated runs and interactive simulations.

**Results:** libRoadRunner is a self-contained library, able to run both as a component inside other tools via its C++ and C bindings, and interactively through its Python interface. Its Python Application Programming Interface (API) is similar to the APIs of MATLAB (**www.mathworks.com**) and SciPy (**http://www.scipy.org/**), making it fast and easy to learn. libRoadRunner uses a custom Just-In-Time (JIT) compiler built on the widely used LLVM JIT compiler framework. It compiles SBML-specified models directly into native machine code for a variety of processors, making it appropriate for solving extremely large models or repeated runs. libRoadRunner is flexible, supporting the bulk of the SBML specification (except for delay and non-linear algebraic equations) including several SBML extensions (composition and distributions). It offers multiple deterministic and stochastic integrators, as well as tools for steady-state analysis, stability analysis and structural analysis of the stoichiometric matrix.

**Availability and implementation:** libRoadRunner binary distributions are available for Mac OS X, Linux and Windows. The library is licensed under Apache License Version 2.0. libRoadRunner is also available for ARM-based computers such as the Raspberry Pi. http://www.libroadrunner.org provides online documentation, full build instructions, binaries and a git source repository.

**Contacts:** hsauro@u.washington.edu or somogyie@indiana.edu

**Supplementary information:** Supplementary data are available at *Bioinformatics* online.

## 1 Introduction

Dynamic network models (Sauro, 2014) of metabolic, gene regulatory, protein-signaling and electrophysiological networks require the specification of components, interactions, compartments and kinetic parameters. The Systems Biology Markup Language (SBML) (Hucka *et al.*, 2003) has become the *de facto* standard for

declarative specification of these types of model (Dräger *et al.*, 2014; Sauro and Bergmann, 2009).

Popular tools for the development, simulation and analysis of models specified in SBML include COPASI (Hoops *et al.*, 2006), Systems Biology Workbench (SBW) (Bergmann and Sauro, 2006), The Systems Biology Simulation Core Algorithm (TSBSC) (Keller *et al.*, 2013), Jarnac (Sauro and Fell, 2000), libSBMLSim (Takizawa *et al.*, 2013), SOSLib (Machné *et al.*, 2006), iBioSim (Myers *et al.*, 2009), PySCeS (Olivier *et al.*, 2005) and VirtualCell (Moraru *et al.*, 2008). Some of these applications are stand-alone packages designed for interactive use, with limited reusability as components in other applications. Few are reusable libraries. Currently, none is fast enough to support emerging applications that require large-scale simulation of network dynamics. For example, multi-cell virtual-tissue simulations (Hester *et al.*, 2011) often require simultaneous simulation of tens of thousands of replicas of dynamic network models residing in their cell objects and interacting between cells. In addition, optimization methods require generation of time-series for tens of thousands of replica networks to explore the high-dimensional parameter spaces typical of biochemical networks (Bouteiller *et al.*, 2015).

We designed libRoadRunner to provide: (i) Efficient time-series generation and analysis of large or multiple SBML-based models; (ii) A comprehensive and logical Application Programming Interface (API); (iii) Interactive simulations in the style of IPython and MATLAB and (iv) Extensibility.

Most existing SBML simulation engines use built-in interpreters to parse and execute SBML model specifications. Interpreted execution is simple and flexible, but much slower than execution of compiled code. Other simulation engines generate compiled executables from SBML by first converting SBML-specified models into a general-purpose-language representation. The engines then call an external compiler to translate the general-purpose-language into an executable shared library to load at run time. E.g., SBW-roadRunner in the SBW suite (Bergmann and Sauro, 2006) converts SBML into C# [see § 1.4 of (Aho *et al.*, 1986)], then compiles the C# using the built-in compiler from the .NET distribution. This approach generates relatively fast executables. However, it requires distribution of a separate compiler or a redistributable runtime, reducing portability.

A more efficient approach to SBML-to-executable compilation uses a specialized *just-in-time* (JIT) compiler, to compile SBML into an optimized Intermediate Language (IL) representation and the IL code into native executable machine code directly in-memory. Ackermann *et al.* (2009) used JIT compilation to generate CUDA code from SBML and execute it on an Nvidia GPU. libRoadRunner and the Stochastic Simulation Compiler (SSC) (Lis *et al.*, 2009) both compile dynamic network-model specifications into executables, SSC focusing on stochastic simulation of rule-based models and libRoadRunner on SBML-specified models. libRoadRunner supports execution of a broad range of SBML models on CPUs using a custom-built JIT compiler [based on the LLVM JIT compiler framework (Lattner and Adve, 2004)] which translates SBML into highly optimized executable code for a broad range of processors. LLVM-based compilers are small, so all JIT operations occur in memory, without external file or compiler access, ensuring fast, self-contained simulations and a relatively small distribution package.

## 2.1 Capabilities

libRoadRunner supports time-course simulation of deterministic and stochastic models. It also supports steady state analysis, stability analysis and structural analysis of the stoichiometry matrix (Reder, 1988). libRoadRunner supports almost the entire SBML L3V1 specification, including hierarchical model composition and the distribution package. Its lacks support only for delay equations and non-linear algebraic rules.

## 2.2 Portability

Because new hardware platforms appear frequently, a modern simulator must be portable. libRoadRunner has no run-time dependencies beyond standard system libraries and it supports any processor LLVM supports. LLVM future-proofs libRoadRunner, ensuring that we need not change the front end of the compiler to support new processor architectures. libRoadRunner is written in C++, so it interfaces easily with other C++-based software. libRoadRunner also provides a C language wrapper for cross-language support and uses SWIG (Beazley, 1996) to provide a customized native-Python API. The use of SWIG will allow future support for additional native language bindings, such as JavaScript, R or Octave, depending on demand.

## 2.3 Extensibility

libRoadRunner's modular design is easy to maintain and extend. All top-level components, such as solvers and integrators, interact via well-defined boundaries (*pure virtual interfaces*) to reduce inter-component dependencies and hide their internal details. A new solver needs only to implement a standard interface to function as part of the library, so adding a solver requires no modification of pre-existing code.

## 2.4 SBML as a declarative language

SBML (Hucka *et al.*, 2003) is a *declarative* specification format for network models. Because of its history, SBML terminology derives from biochemistry and includes common biochemical-reaction abstractions like reaction steps, compartments and reaction rate laws, though it can describe any model of form:

$$\frac{d}{dt}\mathbf{x}(t) = f(\mathbf{x}(t), \mathbf{p}), \tag{1}$$

where $\mathbf{x}$ is the state vector of the model, and $\mathbf{p}$ is a vector of time-independent parameters.

SBML-specified models can also include *events*, discontinuous state changes, which trigger under specified conditions. libRoadRunner correctly handles SBML-specified events and extends the SBML specification by allowing an SBML event to call an arbitrary user-defined function.

Declarative specification languages, like SBML, define component objects and their interactions, rather than defining procedural control flow (i.e. the sequence in which computational operations proceed on execution). An SBML specification lists only the network component objects, their interactions and rate relations and events which change these interactions and rates, all of which are intrinsic abstractions in SBML. Thus, an author writing a model specification in SBML can focus on the underlying biology or chemistry of the model rather than on how to implement the model as a simulation. Because SBML does not specify the computational operations to implement a model, the control flow, the solvers to use, or how to store the model's elements, an SBML compiler or interpreter must generate them appropriately from the SBML specification. Thus, compiling an SBML model specification is more complex than compiling a functionally equivalent model specification in a procedural language.

SBML model specifications are easier to share than procedural specifications of equivalent models because they are not implementation dependent; any of the numerous SBML compliant tools can process any SBML model specification. This portability allows model archiving (e.g. in exchange repositories such as BioModels (Le Novere *et al.*, 2006) and reuse and the relatively simple assembly of multiple SBML-specified sub-models into larger models. It also simplifies the scientific validation of SBML-specified models and ensures that SBML-specified models remain usable, even if the specific software tools that generated them fall out of use.

## 2 Architecture

libRoadRunner is a self-contained, easily embedded library with an object-oriented API natively accessible in C, C++ and Python (SWIG allows easy extension to other languages). libRoadRunner's *component-oriented design* specifies a small number of standardized software interfaces (protocols) and how they interact, implemented using standard C++ data types. Component-orientation separates the implementation of a component from its interface, so components are easy to add or replace and component swapping requires no changes to existing code. E.g., we can add new integrators, steady-state solvers or SBML compilers to the libRoadRunner library via the `Integrator`, `SteadyStateSolver` and `ExecutableModel` interfaces, respectively. libRoadRunner includes three implementations of the `Integrator` interface: two deterministic integrators [one based on the CVODE integrator from the Sundials suite (Hindmarsh *et al.*, 2005) and the other a standard fourth-order Runge–Kutta method] and a standard Gillespie Direct Method SSA stochastic integrator (Gillespie, 1977). libRoadRunner implements the `SteadyStateSolver` interface as a class which uses the NLEQ (Nowak and Weimann, 1991) solver, and we are currently developing additional methods. libRoadRunner implements the `ExecutableModel` interface as a class which uses our SBML-to-CPU JIT compiler (see § 3). libRoadRunner statically links to the third-party libraries LLVM (Lattner and Adve, 2004), libSBML (Bornstein *et al.*, 2008), CVODE, NLEQ2, LAPACK (http://www.netlib.org/lapack/) and POCO (http://pocoproject.org/).

## 3 SBML-to-CPU-executable compilation

LibRoadRunner's SBML JIT compiler compiles SBML models in the form of strings to executable native machine code, in memory. Compilation follows the canonical compiler phases (Aho *et al.*, 1986): (i) lexical analysis, (ii) syntactic analysis, (iii) semantic analysis, (iv) intermediate code generation, (v) code optimization and (vi) native code generation. Standard generic libraries can perform phases 1, 2, 5 and 6. However, semantic analysis (phase 3) is specific to the source language.

In phases 1 and 2, the compiler reads the source text, parses it, and extracts and converts the text's syntactic information into an *abstract syntax tree* (AST) data structure. Each node in the AST is an *essential construct* such as an operator, symbol, literal or function call. Most SBML simulators use components of the libSBML (Bornstein *et al.*, 2008) library to perform lexical and syntactic analyses of SBML model specifications.

In phases 3 and 4, the compiler reads the AST and assembles it into a sequence of IL (IL, a machine-independent assembly language) instructions, which form a procedural instantiation of the SBML model specification.

CPUs cannot execute IL programs directly, so phases 5 and 6 optimize the IL (by removing redundant operations, optimizing memory layout, ...) and convert it into executable machine code. libRoadRunner uses components of the LLVM library for phases 5 and 6.

After the completion of phases 1–6, the JIT compiler returns the executable code in the form of a list of callable functions to the calling program.

During phase 3 (semantic analysis), the compiler must map language symbols to memory address locations. The compiler of a procedural language, such as C, allocates a memory location to each symbol (e.g. a variable or function declaration), and resolves that symbol to that location whenever the source code references that symbol. Procedural-language compilers map symbols to memory locations using a *symbol table* data structure. SBML has no construct for creating new variables or eliminating variables at run-time, so the compiler can compute the exact memory requirements for all symbols and store the symbols in a contiguous memory block. At run-time, during a time-series computation, the libRoadRunner library connects a JIT-compiled function to an integrator, which, in turn, calls a function which calculates the rate of change of the state vector. Because both the state vector and the rate of change occupy contiguous memory blocks and have the same layout as the SBML model variables, the calls pass only two pointers and require no memory copying or rearrangement.

However, compilation of SBML poses challenges. SBML model specifications may define rules which state that an expression should replace a specified symbol, or a rate rule which specifies a rate of change of the value of a symbol, rather than the symbol value itself. SBML also allows different rules to apply in different contexts, such as special rules which only apply when the model is loaded (initial assignment rules). Mapping symbol names to memory locations is not one-to-one so a symbol table is insufficient to store the mapping.

Some SBML model simulators allocate storage space for both normal and rule-defined symbols and use auxiliary functions to evaluate the rules at run-time as the symbols are read. However, this approach wastes memory storing symbols which resolve to other symbols and complicates execution, as the run-time must keep track of rule dependencies.

Our solution is to extend the symbol table into a *symbol forest*, a hash table which maps symbol names to ASTs describing all the symbols' rules. The SBML compiler uses the symbol forest much as a procedural-language compiler uses a symbol table, to resolve symbol names to memory locations. However, the symbol forest must apply any rules which relate symbols to determine the memory location for a given symbol. E.g., if the symbol $x$ has the assignment rule $x \rightarrow y + 1$, whenever the compiler references symbol $x$, the symbol forest will find the rule, generate a sequence of IL instructions which both implement the right hand side (RHS) of the rule and create a temporary variable to store the result of the rule calculation. The symbol forest then stores this sequence of IL instructions and returns the memory location of the instruction sequence to the compiler. Later in compilation, the LLVM code generator translates these IL instructions into an executable, which calculates and returns the value of the symbol at run-time. The symbol forest resolves automatically recursive rules in which the symbols in the RHS of a rule depend on other rules.

Naïvely generating IL expansions of the rule definitions inline and creating temporary variables for rule evaluation would generate redundant instructions which would slow both compilation and execution. libRoadRunner's *scoped symbol cache* reduces such redundancy. Many functions in libRoadRunner do not modify

SBML-defined parameters and variables during function execution, so any rules depending on these parameters and variables need evaluation only once during a given call to these functions. Even if the rule involves a condition, e.g. $x \rightarrow \{b \text{ if } (a > 1) \text{ else } c)$, if the function does not change the values of $a$, $b$ and $c$, the function needs to evaluate the rule to obtain the value of $x$ only once per call. The SBML compiler therefore generates code which evaluates the rule whenever the function is called and stores the result in a temporary variable. During a call to the function, the first reference to the symbol evaluates the rule and caches its result, and any subsequent references to that symbol during the function call reference the cached value. Using a scoped symbol cache reduces memory usage and execution time, typically by a factor of 10 for large models.

When JIT-compiled functions contain conditional branches which contain rules, the SBML compiler generates redundant IL code, which slows compilation (which scales as the size of the IL code) but has no speed cost at execution. If the compiler examined all possible branches, determined what rules were present, and created temporary variables to contain the results of the rule evaluations, it would reduce the size of the resulting IL code, speeding compilation. However, slower execution would offset the faster compilation, since the executable would evaluate all rules in all branches, not only those which it needed. We may add a compiler directive to allow the user to choose the second option in a future release of libRoadRunner.

## 4 Results

### 4.1 Performance

Simulation engines which interpret SBML models (Romer *et al.*, 1996), are inherently slower, sometimes much slower, than engines which generate and execute complied code. libRoadRunner uses JIT compilation to generate particularly fast simulations.

We benchmarked libRoadRunner and Jarnac (Sauro and Fell, 2000), a popular interpreter-based network simulator, for a variety of network model types (Table 1; Supplementary Materials Table S1). libRoadRunner's faster execution speed is particularly evident when solving large models, such as BIOMOD14 (Table 1), a mass-action model including a large number of states. We also checked the scaling of the execution time ($t$) in the number of replicas ($N$) of a Brusselator model, approximating the use of libRoadRunner in a virtual-Tissue simulation with thousands of cells, with each cell including its own replica of an SBML-specified network model. The run time for libRoadRunner scales as $t \sim N$, whereas the run time for Jarnac scales as $t \sim N^{2.6}$ (Supplementary Materials Fig. S2). Thus libRoadRunner is more suitable than Jarnac for use in Virtual-Tissue simulations or other simulations requiring many replicas of one or more networks. The Supplementary Materials present the full benchmark comparisons.

**Table 1.** Ratios of Jarnac and libRoadRunner run times and total execution times (including loading) for selected network models (Supplementary Materials for full benchmark data)

| Model name | Run time Jarnac/ libRoadRunner | Total time Jarnac/ libRoadRunner |
|---|---|---|
| Jana wolf | 4.30 | 2.08 |
| BIOMOD14 | 311 | 3.98 |
| BIOMOD33 | 3.14 | 0.35 |
| Brusselator500 | 22 875 | 225 |

Simulation speed depends on the performance of both the state-vector rate calculation and the numeric integrator. Because we cannot separate these calculations in most SBML-model packages, we also compared an SBML model JIT-compiled using libRoadRunner with a hard-coded C++ version of the same model. The model implemented 1000 instances of a Hofmeyr–Cornish-Bowden unimolecular reaction, in which a single substrate reversibly goes to a single product ($S \rightarrow P$) at a rate of (Hofmeyr and Cornish-Bowden, 1997; Sauro, 2012):

$$\frac{\mathrm{Vm}\left(\frac{S}{\mathrm{Km}_1}\right)\left(1-\frac{\Gamma}{K_{eq}}\right)\left(\frac{S}{\mathrm{Km}_1}+\frac{P}{\mathrm{Km}_2}\right)^{h-1}}{\frac{1+(M/k)^h}{1+\sigma(M/k)^h}+\left(\frac{S}{\mathrm{Km}_1}+\frac{P}{\mathrm{Km}_2}\right)^{h}}.$$

On a 64-bit Linux system, using the clang C++ compiler, execution of 1000 to 15 000 time steps using the JIT-compiled SBML model and the hard-coded C++ specification took the same time, showing that the flexibility of libRoadRunner does not entail any significant speed cost.

### 4.2 Python bindings

libRoadRunner's Python API employs a simple, concise object model, and follows the style and conventions of the widely used SciPy library for ease of learning. The API provides high performance, low-overhead access to the libRoadRunner library. The API only communicates using standard Python data types such as lists, dictionaries and NumPy arrays, which simplifies integration with existing applications. The NumPy array type is a data structure which wraps a Python interface around a standard C numeric array. Even large NumPy arrays have low overhead, since they return only pointers to internal arrays owned by the libRoadRunner library, with no copying of memory.

To provide the functionality of the Pandas (http://pandas.pydata.org) *DataFrame* object, libRoadRunner extends the NumPy array to contain row and column name information, to support access to rows and columns by name, and to format this name information for console output. Unlike the Pandas DataFrame, which replaces the Numpy array and requires conversion to work with Python and Numpy functions, the libRoadRunner array is a standard Numpy array which any SciPy function can use. The libRoadRunner array requires only a single line to display the components and interaction names in the stoichiometry matrix:

```
print(r.getFullStoichiometryMatrix())
      J0,   J1,   J2,   J3,   J4
S1  [[1,   -1,    0,    0,    0]
S2   [0,    1,   -1,    0,    0]
S3   [0,    0,    1,   -1,    0]
S4   [0,    0,    0,    1,   -1]].
```

Running a libRoadRunner simulation only requires loading a model and calling a simulation method. Defaults preset the time spans and number of points a simulation generates. By default, the simulate method returns time in the first column and all floating model species in additional columns:

```
r = RoadRunner("glycolysis.sbml")
m = r.simulate(plot = True)
```

Here `m` is a NumPy array, and the optional `plot = True` argument to the `simulate` method calls the standard plotting library, matplotlib, to display a basic time-series plot of the simulation

results. Optional arguments can customize the simulation, e.g. to generate a 100 data-point time series for parameter 'p' and concentration 'S1' from an SBML-specified model between times $t = 0$ and $t = 12$, we specify:

```
r = RoadRunner("glycolysis.sbml")
m = r.simulate(0, 12, 100, ['time', 'p', '[S1]'])
```

A variety of other built-in symbols access reaction rates, rates of change, eigenvalues, etc. Like a MATLAB top-level function, the libRoadRunner `simulate` method provides a consistent front end to all libRoadRunner's integration engines. Because MATLAB is familiar to many scientists, the MATLAB-like architecture reduces the effort to learn the libRoadRunner API. To simplify generation of simulation documentation, libRoadRunner methods support internal pydoc strings, which interactive Python environments such as IPython or Tellurium (http://tellurium.analogmachine.org/) make available as pop-up hints.

The libRoadRunner API uses dynamic Python object properties to simplify access to SBML model values. Loading an SBML-specified model via libRoadRunner automatically adds the SBML model's symbol names to the `RoadRunner` object, allowing dynamic introspection and modification of the object. If a model contains parameters and species `'x', 'y', 'S1', 'S2'`, the RoadRunner object will include these names as properties, which a user can read or set. E.g.,

```
# load a model that has ids 'x', 'y' and 'S1'
r = RoadRunner('some_model.xml')
r.x = 1.5 # set the 'x' parameter to 1.5
r.y = 2.0 # set the 'y' parameter to 2.0
print(r.S1) # print the 'S1' species concentration
```

### 4.3 Support for analysis

The C# roadrunner package inspired libRoadRunner, which inherits many of roadrunner's analysis functions, including: methods to calculate scaled and unscaled control coefficients, elasticities, sensitivity to changes in all parameters, including conserved quantities, eigenvalues and eigenvectors and stoichiometric quantities like the Link and K matrices (Reder, 1988). libRoadRunner can also compute frequency responses to generate Bode plots.

### 4.4 Identification of conserved quantities

Many biochemical network computations require identification of conserved quantities (*moieties* in biochemical usage) and elimination of linearly dependent species to avoid inversion of singular Jacobian matrices (Vallabhajosyula *et al.*, 2006). libRoadRunner implements a libSBML plug-in which performs this reduction on SBML Document objects, first identifying conserved quantities and dependent species, then adding the conserved quantities to the document as set of global parameters and replacing the dependent species with assignment rules. The user can modify these conserved quantities, which behave as parameters, to investigate their effect on the dynamics of the model.

## 5 Use cases

libRoadRunner's ease of use, ability to handle complex SBML models and fast model execution speed have led to its rapid adoption in a variety of applications.

### 5.1 The tellurium interactive network solver

Tellurium is a cross-platform integrated Python environment based on the Spyder IDE (http://code.google.com/p/spyderlib/). Tellurium combines libRoadRunner, libSBML, Antimony (Smith *et al.*, 2009), libSEDML (http://libsedml.sourceforge.net/libSedML) and other packages to provide a comprehensive development and analysis environment for Antimony-specified models. libRoadRunner's concise syntax and intuitive Python API are essential to Tellurium's support for interactive creation, simulation and analysis of dynamic network models.

### 5.2 Integrating SBML-model specifications into multi-cell virtual-tissue models simulated in CompuCell3D

CompuCell3D (CC3D), a simulation environment for multi-scale, multi-cell virtual-tissue model development and simulation, was the first tool to adopt libRoadRunner as a core engine. CC3D defines a *cell* object class and *behavior* methods to allow cell objects to grow, divide, die, secrete/absorb chemicals, move, etc.... libRoadRunner integration with CC3D allows the state of an SBML-specified model inside a cell object to control the CC3D parameters describing the cell object's behaviors, and *vice versa*.

E.g., in a model of changes in cell–cell adhesion leading to invasive tumor phenotypes, the CC3D cell objects have a CC3D parameter *adhesion-molecule density*, which controls the CC3D behavior *cell–cell adhesion*. An SBML-specified model relates the level of the transmembrane adhesion receptor E-cadherin in each cell to the cells' level of $\beta$-catenin (Andasari *et al.*, 2012). The CC3D-model specification uses the libRoadRunner Python API to connect the CC3D *adhesion-molecule density* to the SBML-model's transmembrane E-Cadherin level. At run-time, libRoadRunner time evolves the network models inside cells, while a specialized CC3D engine handles the evolution of the cell objects.

Another use of SBML models in virtual-tissue modeling is simulation of Delta-Notch patterning during embryonic development. Delta and Notch are heterophilic transmembrane receptors whose signaling is mutually inhibitory within a cell. The level of signaling depends on both the amount of Delta on the membrane of a cell and the amount of Notch on the surfaces of neighboring cells and *vice versa*. Thus, the dynamics of the signaling network depends not only the model within the cell, but the cell's pattern of contacts with neighboring cells and their levels of Delta and Notch. To model this situation, we create CC3D cell objects and arrange them in an *epithelium* (a quasi-2D sheet). Each cell contains an SBML-specified model that describes how the cell's levels of membrane-bound and cytosolic Delta and Notch change, for a particular input level of transmembrane Delta and Notch signaling (Swat *et al.*, 2012). A Python layer uses the libRoadRunner API to calculate the strength of Delta and Notch signaling each cell experiences from the amount of Delta on the membrane of each cell, the amount of Notch on the membrane of each adjacent cell (adjacency is a CC3D model parameter) and the CC3D model's area of contact between each pair of cell neighbors. libRoadRunner then updates cells' Delta–Notch signaling and regulatory networks using these signaling strengths as boundary conditions, while CC3D updates the cell shapes, positions adjacencies and contact areas. Together, these interactions produce the checkerboard pattern typical of embryonic Delta–Notch signaling.

### 5.3 Multi-scale virtual-tissue modeling of liver metabolism

The Virtual Liver Network has developed an organ-level model of human galactose clearance which includes single-cell metabolism of

hepatocytes, the ultra-structure and micro-circulation of hepatic tissues, and the structure of the entire organ (https://github.com/matthiaskoenig/multiscale-galactose).

The liver model includes an SBML-specified model of the *sinusoid*, the smallest functional unit of the liver, consisting of a perfused capillary surrounded by hepatocytes. This model contains a biochemical network describing galactose metabolism in individual hepatocytes. Coupling via SBML-specified discretized transport equations for convection and diffusion results in a model with several thousand components and interactions. The sinusoid model uses SBML events to describe the time-varying supply of galactose to the liver. Accounting for heterogeneity in blood flow and tissue architecture requires simulation of more than $2 \times 10^5$ replicas of the model with varying tissue and flow parameters. This number of replicas was feasible because of libRoadRunner's fast time-series generation and support for variable step sizes, which dramatically reduced output file size. Using the CVODE solver, single simulation runs of the liver model take around 5–7 s on RoadRunner, resulting in a total simulation time of 4 h for $10^5$ simulations on a cluster with 40 cores. libRoadRunner's Python API supported rewrite-free integration of the SBML models into a complex pre-existing modeling workflow, which included data management using Django, model annotation using Python bindings to libSBML, model prototyping using Python bindings to Antimony and visualization of results using the Python REST interface to Cytoscape (Shannon *et al.*, 2003) with CySBML and CyFluxViz.

### 5.4 Modeling of synaptic, neuronal and neuron network dynamics in the MEMORY platform

The MEMORY platform [Multi-scale intEgrated Model Of the neRvous sYstem, formerly EONS (Bouteiller *et al.*, 2008)] simulates the function and dynamics of elements ranging from single channels or receptors (*elementary models*), to synapses, which include many elementary models, to neurons, which themselves may include a large number of synapses. MEMORY depends on libRoadRunner's flexibility and ease of use to assemble such complex hierarchical models. E.g., an SBML-specified neuron model may include many SBML-specified synapse models, each of which includes many SBML-specified neurotransmitter release and diffusion, AMPA receptor and NMDA-receptor models (both ionotropic receptors for the glutamate neurotransmitter). Neuronal models may be large, e.g. representing 10 ionotropic synapses in a CA1 neuron model (Izhikevich, 2003) requires 73 events, 290 reactions, 414 rules and 1459 parameters, so libRoadRunner's fast time-series generation is essential for MEMORY to solve complex neuronal models quickly.

To ensure that a neuronal model quantitatively predicts biological functions like membrane potentials or intracellular molecular concentrations, MEMORY can optimize the model's parameters by fitting between multiple simulation and experimental time-series for characteristics including changes in receptor conductance, desensitization properties and spiking patterns. MEMORY uses evolutionary multi-objective optimization [from the EMOO framework (Bahl *et al.*, 2012)], which requires large numbers of simulation replicas. E.g., elementary-model optimization of an NMDA-receptor model with respect to eight distinct experimental results for dynamical changes in receptor-channel conductance following paired-pulse stimulation, required 15 000 generations with 400 individuals per generation, i.e. 6 million simulation replicas (corresponding to 13 000 h of simulated time). libRoadRunner took 66 h to run the entire optimization on a 400-node computer cluster, orders of magnitude faster than other SBML simulators (Bouteiller *et al.*, 2015).

## 6 Conclusions

libRoadRunner's speed and ease of integration allow researchers to solve very large models, include models embedded in multi-scale systems and run large ensembles of smaller models. libRoadRunner's Python API makes simulations easy to learn, while its C++ and C APIs are attractive to developers wishing to integrate libRoadRunner capabilities into existing simulation frameworks. libRoadRunner runs on ×86 and ARM architectures and Windows, Mac OS X, Linux, Raspberry Pi, NVIDIA Jetson TK1 and ADAPTEVA Parallella boards. libRoadRunner's speed and ARM support will make tablet-based network applications practical despite tablets' relatively slow CPU speeds. libRoadRunner's support for inexpensive processor boards such as the Raspberry Pi-2 allows individual researchers and students to more easily study cluster parallelization options.

## 7 Future work

*Improve Steady-State Solvers* libRoadRunner uses the FORTRAN NLEQ2 non-linear steady-state solver, which is not thread safe. Exclusive access locks (*mutexes*) are on the NLEQ solver which restricts its use to one thread at a time. To eliminate this restriction, we plan to add several thread-safe steady-state solvers.

*Extensions* A suite of extensions to libRoadRunner is under development. They include a bifurcation extension and a set of parameter optimizers.

## Funding

E.T.S., M.H.S. and J.A.G. acknowledge support from NIH grants R01 GM077138, U01 GM111243, R01 GM076692 and EPA RD83500101. M.K. acknowledges support from the Federal Ministry of Education and Research (BMBF, Germany) within the Virtual Liver Network (VLN grant 0315741). J.M.B. acknowledges support from NIH grants P41 EB001978 and U01 GM104604. H.M.S. acknowledges support from NIH grant R01 GM081070. The content is solely the responsibility of the authors and does not necessarily represent the views of the National Institutes of Health. We acknowledge Totte Karlsson for the original C# to C++ translation, C compiler backend and C API, Stanley Gu for testing the library as a web service, Lucian Smith for developing part of the test suite, Michael Galdzicki for writing detailed build instructions and testing for developers. H.M.S. conceived the project and helped with documentation, design and testing. E.T.S. designed the overall architecture of the libRoadRunner, developed the LLVM-based JIT compiler and wrote documentation. We thank Wilbert Copeland for bug fixing, testing and redesigning the integrator interface. M.H.S. conducted Linux testing and builds, M.K., J.A.G. and J.M.B. tested the code in simulations. KM ported the code to ARM processors and carried out the performance testing. We thank Holly Sawyer for proofreading the final draft.

*Conflict of Interest*: none declared.

## References

Ackermann,J. *et al.* (2009) Massively-parallel simulation of biochemical systems. *GI Jahrestagung*, In: *Proceedings of Massively Parallel Computational Biology on GPUs*, pp. 739–750.

Aho,A. *et al.* (1986) *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Reading, Massachusetts.

Andasari,V. *et al.* (2012) Integrating intracellular dynamics using CompuCell3D and Bionetsolver: applications to multiscale modelling of cancer cell growth and invasion. *PLoS One*, **7**, e33726.

Bahl,A. *et al.* (2012) Automated optimization of a reduced layer 5 pyramidal cell model based on experimental data. *J. Neurosci. Methods*, **210**, 22–34.

Beazley,D.M. (1996) SWIG: an easy to use tool for integrating scripting languages with C and C++. In: *Proceedings of the 4th USENIX Tcl/Tk Workshop, USENIX Association, Berkeley, California*, pp. 129–139.

Bergmann,F.T. and Sauro,H.M. (2006) SBW—a modular framework for systems biology. In: *WSC '06 Proceedings of the 38th Conference on Winter Simulation*, Association for Computing Machinery (ACM), New York City, New York, pp. 1637–1645.

Bornstein,B.J. *et al*. (2008) LibSBML: an API library for SBML. *Bioinformatics*, **24**, 880–881.

Bouteiller,J.-M.C. *et al*. (2008) Modeling glutamatergic synapses: insights into mechanisms regulating synaptic efficacy. *J. Integr. Neurosci.*, **7**, 185–197.

Bouteiller,J.-M.C. *et al*. (2015) Maximizing predictability of a bottom-up complex multi-scale model through systematic validation and multi-objective multi-level optimization. In: *Proceedings of the 7th International IEEE/EMBS Conference on Neural Engineering (NER)*, IEEE Engineering in Medicine and Biology Society (EMBS), Piscataway, New Jersey.

Dräger,L. *et al*. (2009) Improving collaboration by standardization efforts in systems biology. *Front Bioeng. Biotechnol.*, **2**, 61.

Gillespie,D.T. (1977) Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, **81**, 2340–2361.

Hester,S.D. *et al*. (2011) A multi-cell, multi-scale model of vertebrate segmentation and somite formation. *PLoS Comput. Biol.*, **7**, e1002155.

Hindmarsh,A.C. *et al*. (2005) SUNDIALS: suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw. (TOMS)*, **31**, 363–396.

Hofmeyr,J.-H.S. and Cornish-Bowden,A. (1997) The reversible hill equation: how to incorporate cooperative enzymes into metabolic models. *Comput. Appl. Biosci. (CABIOS)*, **13**, 377–385.

Hoops,S. *et al*. (2006) COPASI—a complex pathway simulator. *Bioinformatics*, **22**, 3067–3074.

Hucka,M. *et al*. (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, **19**, 524–531.

Izhikevich,E.M. (2003) Simple model of spiking neurons. *IEEE Trans. Neural Netw.*, **14**, 1569–1572.

Karr,J.R. *et al*. (2012) A whole-cell computational model predicts phenotype from genotype. *Cell*, **150**, 389–401.

Keller,R. *et al*. (2013) The systems biology simulation core algorithm. *BMC Syst. Biol.*, **7**, 55.

Lattner,C. and Adve,V. (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: *IEEE International Symposium on Code Generation and Optimization (CGO)* 2004. IEEE, pp. 75–86.

Le Novere,N. *et al*. (2006) Biomodels database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Res.*, **34**(Suppl. 1), D689–D691.

Levenberg,K.A. (1944) Method for the solution of certain non-linear problems in least squares. *Q. Appl. Math.*, **2**, 164–168.

Lis,M. *et al*. (2009) Efficient stochastic simulation of reaction–diffusion processes via direct compilation. *Bioinformatics*, **275**, 2289–2291.

Machné,R. *et al*. (2006) The SBML ODE solver library: a native API for symbolic and fast numerical analysis of reaction networks. *Bioinformatics*, **22**, 1406–1407.

Marquardt,D.W. (1963) An algorithm for least-squares estimation of nonlinear parameters. *J. Soc. Ind. Appl. Math.*, **11**, 431–441.

Moraru,I.I. *et al*. (2008) Virtual cell modelling and simulation software environment. *IET Syst. Biol.*, **2**, 352–362.

Myers,C.J. *et al*. (2009) iBioSim: a tool for the analysis and design of genetic circuits. *Bioinformatics*, **25**, 2848–2849.

Nowak,U. and Weimann,L. (1991) A family of Newton Codes for Systems of Highly Nonlinear Equations. Technical Report TR-91-10, ZIB, Konrad-Zuse-Zentrum fr Informationstechnik Berlin. ZIB Technical Report.

Olivier,B.G. *et al*. (2005) Modelling cellular systems with PySCeS. *Bioinformatics*, **21**, 560–561.

Reder,C. (1988) Metabolic control theory: a structural approach. *J. Theor. Biol.*, **135**, 175–201.

Reynolds,D.R. *et al*. (2014) ARKode: a library of high order implicit/explicit methods for multi-rate problems. In: SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pennsylvania.

Romer,T.H. *et al*. (1996) The structure and performance of interpreters. *ACM SIGPLAN Notices*, **31**, 150–159.

Sauro,H.M. (2012) *Enzyme Kinetics for Systems Biology, 2nd edn*. Ambrosius Publishing, Seattle, Washington.

Sauro,H.M. (2014) *Systems Biology: An Introduction to Pathway Modeling*. Ambrosius Publishing, Seattle, Washington.

Sauro,H.M. and Bergmann,F.T. (2009) Software tools for systems biology. In: Liu,E.T. and Lauffenburger,D.A. (eds.) *Systems Biomedicine*. Academic Press, Waltham, Massachusetts, pp. 289–312.

Sauro,H.M. and Fell,D.A. (2000) Jarnac: a system for interactive metabolic analysis. In: *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*. Stellenbosch University Press, Western Cape, South Africa, pp. 221–228.

Shannon,P. *et al*. (2003) Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res.*, **13**, 2498–2504.

Smith,L.P. *et al*. (2009) Antimony: a modular model definition language. *Bioinformatics*, **25**, 2452–2454.

Swat,M.H. *et al*. (2012) Multi-scale modeling of tissues using CompuCell3D. In: Elsevier,B.V. (ed.), *Methods in Cell Biology*. Elsevier, Amsterdam, Netherlands, pp. 325–366.

Takizawa,H. *et al*. (2013) LibSBMLSim: a reference implementation of fully functional SBML simulator. *Bioinformatics*, **29**, 1474–1476.

Vallabhajosyula,R.R. *et al*. (2006) Conservation analysis of large biochemical networks. *Bioinformatics*, **22**, 346–353.