R as an Integration Tool in High Performance Computing – Lessons Learned

Patrick Boba, Stefan Gries, Kay Hamacher

Computational Biology and Simulation
Technische Universität Darmstadt
Schnittspahnstraße 2
64283 Darmstadt
Germany
{boba,hamacher}@cbs.tu-darmstadt.de

Abstract: In this paper we discuss the usage of the statistical software R to integrate software components for large-scale simulations. To this end, we show how to interface different components written in various programming languages under the umbrella of R.

We base our study on well-known tools such as COPASI [HSG+06] and connect it to our own R package SOMNIBIEN (Simulation Of Metabolic Networks Influenced By Internal And External Noise). In particular, we show the need to use Fortran code for fast simulation via stochastic differential equations. This implies the need to interface with code in C, while offering the opportunity to leverage the superior analysis capabilities of R and its plotting capabilities.

Eventually, our system compiles models defined in an XML-dialect, translates them into C in the background and executes a Fortran solver for stochastic differential equations (SDEs) on them. Furthermore, we present the integration with download-capabilities from the XML-based BioModels Database [FH03a].

For the lessons learned, we describe pitfalls and performance bottlenecks and provide guidelines for future work on the integration of different code repositories into one R framework. Finally, we show performance evaluations on multi-core parallelization within our package.

1 Introduction

Nowadays network simulations play a major role in biological sciences and beyond. Especially in systems biology, simulations of genetic or metabolic networks have become a state-of-the art method to analyze properties and dynamics of those networks [KLW+09] and assign evolutionary meaning to the experimentally determined network topologies. While network science has identified several domains in which network simulations are an important research tool, here we will focus on biological, intracellular networks to illustrate our lessons. Here, computational biology poses an interesting challenge due to several problems plaguing the field: legacy code infrastructure, several often contradicting programming styles, poorly curated data sources with the need to manually interfere in

models and software, as well as hypothesis driven investigation approaches that demand both HPC resources and visual/interactive analytics.

In general, the network or graph topologies mentioned above can be represented in so-called flow charts (see Fig. 1). Here, vertices are molecular species quantified by their respective concentrations and the edges represent (enzyme driven) chemical reactions.

Changes in the molecular concentrations (A to E in Fig. 1) are characterized by rate constants controlling the fluxes (v_n) between the different chemical species. In the most simple case the stoichiometry of a reaction indicates that one compound is transformed into another compound with a specific reaction rate (see Eq. 1).

$$A \xrightarrow{v_1} B \tag{1}$$

$$\frac{\mathrm{d}A}{\mathrm{d}t} = -v_1 \quad \text{and} \quad \frac{\mathrm{d}B}{\mathrm{d}t} = v_1 \tag{2}$$

Each chemical reaction can be written as an ordinary differential equation (ODE) resulting from the stoichiometry of the involved components. For the example of Eq. 1 we derive Eq. 2 which states that metabolite A's concentration is reduced with speed v_1 and therefore at the same speed metabolite B is produced from A.

In general, stoichiometric coefficients c_{ij} exist for every metabolite M_i and every jth reaction with respective rate v_j. Then, we can write a coupled system of equations as follows:

$$\frac{\mathrm{d}M_i}{\mathrm{d}t} = \sum_{j=1}^{r} c_{ij} v_j \tag{3}$$

Typically, the right-hand side is more complex than in this simple, linear example. The general model is given as *coupled, non-linear* ODEs for which only numerical solutions can be obtained.

Now, this modeling approach is based on the so-called "perfect mixing" paradigm, which assumes that every metabolite and every other molecule is present in sufficient concentrations and more or less homogeneous throughout the cellular volume. But in reality, biology is driven by noise. Therefore, a more detailed model is given by a set of stochastic differential equations (SDEs). The noise then captures effects on the individual reactions like changes in the organisms environment (e.g., temperature, stress, etc.). We therefore need to simulate *and* analyze the output for an SDE system:

$$\mathrm{d}M_i(t) = f_i\left(M_1(t), \ldots, M_r(t), v_i^{\downarrow}, v_i^{\uparrow}\right)\mathrm{d}t + \beta \cdot M_i \cdot \mathrm{d}B_t^{(i)} \tag{4}$$

where $v_i^{\downarrow}, v_i^{\uparrow}$ are the reaction rates of the particular forward and backward reactions for the metabolite i. Furthermore, β scales the strength of the noise and $B_t^{(i)}$ is a Brownian process. Note that i.i.d. noise[1] with $B_t^{(i)} \neq B_t^{(j)}$ for $i \neq j$ as well as universal noise[2]

[1] e.g. to capture deviation from perfect mixing

[2] e.g. for temperature changes throughout the cell

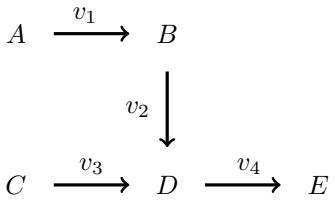

Figure 1: Reaction scheme of a metabolic network represented as a flow chart. The change in concentration of the metabolites indicated by uppercase letters (A to E) is controlled by the corresponding reaction kinetics (v_1 to v_4).

$B_t^{(i)} := B_t$ [GGJ+11] are included in this model of multiplicative noise. Such systems can then be engineered, e.g., optimized in, e.g., its accumulated output by heuristics [Ham13, Ham06] in biotechnology.

A typical work-flow in the life science, includes several manual, error-prone steps: downloading, local storage of model files, simulations by GUI-based tools without parallelization techniques, and final analysis with additional software. Such a work-flow makes it quite hard to achieve desirable goals: 1) reproduciblity is hindered due to the manual nature of file/directory-based documentation, 2) HPC-facilities cannot be leveraged for large-scale parameter scans and optimization efforts, and 3) efficiency is reduced due to a back-and-forth between analysis and simulation programs.

2 Our Contribution

Our primary goal was to develop an `R` package that provides users with a flexible and efficient code base to solve SDEs and analyze the results without migrating between different software systems and architectures. We contribute a) a software package [BGH14] and b) the lessons learned during the integration of the following components into a self-contained, both HPC-enabled and quasi-interactive simulation system:

- download capabilities for curated models – mostly to enable *reproduciblity* of simulations by others
- an `R`-integrated compilation facility of SBML[3]-models to C source code
- an *efficient* SDE solver including a global noise term based on Fortran90 [Ste08].
- interactive sessions via REPL[4]-based compilation and object-file linking mechanisms for the to-be-simulated systems, and

[3] SBML=Systems Biology Markup Language, an XML variant for models from established libraries [Huc03, sbm]

[4] REPL=Read-eval-print loop

- analysis routines in `R` typically employed in time series analysis, e.g., a parallel computation of the time-lagged cross-correlation functions between concentrations of different chemical species.

3 Approach

Our ultimate goal is the simulation, optimization, and investigation towards parameter sensitivity of bio-networks. A very versatile and powerful system to analyze time series and other (big) data is the `R` environment for statistical computing.

Furthermore, wet lab biologists and biotechnologists tend to investigate their systems interactively. Additionally, the involved parameter spaces are high-dimensional and thus demand at times the computational resources only offered by HPC infrastructures. Therefore, we decided to integrate all steps into one system that can be used either in batch mode for intensive simulations or interactively. A typical workflow is shown in Fig. 2.

Note that the major challenge we faced lies in the efficient and transparent integration of various external tools and data sources, while maintaining the possibility of user- and problem-specific analysis procedures. At the same time, the system is supposed to handle *any* SBML model and should not work only for a particular one.

Taken together, this lead us to use COPASI [HSG+06] as an SBML-to-C compiler (run in batch mode), SDERK [Ste08] as the numerical engine for the integration of the SDEs of Eq. 4; in particular to simulate SDEs rather than the less realistic models based on ODEs, C routines to interface components, OpenMP as a parallelization approach, and an R wrapper to steer compilation, linking, and calling of routines.

To give an impression of the achieved simplicity and abstraction of the integrated tools, we show 1 an example workflow in listing. The major steps are the download of the required network model from the database (`getModel`) followed by the creation of the shared object(`PrepareODEFile`). After extracting the network's initial metabolite concentrations and parameters, the simulation can be started at those default values with the `StartSim` command. A common analysis would, for example, continue with the cross correlation function (ccf) of the different metabolite time series.

3.1 Related Work

Besides COPASI [HSG+06], there exist several other (simple) simulation packages such as CellDesigner [Fun03], which is a GUI driven editor that allows editing of existing networks or building them from scratch based on SBML. Various other methods exist to give researchers without a simulation background the tools to simulate intracellular networks, e.g., PySB [LMBS13].

While all these systems address some of the challenges and fulfill some of the requirements from above, we lacked an integrated package – in particular one that interfaces with `R`.

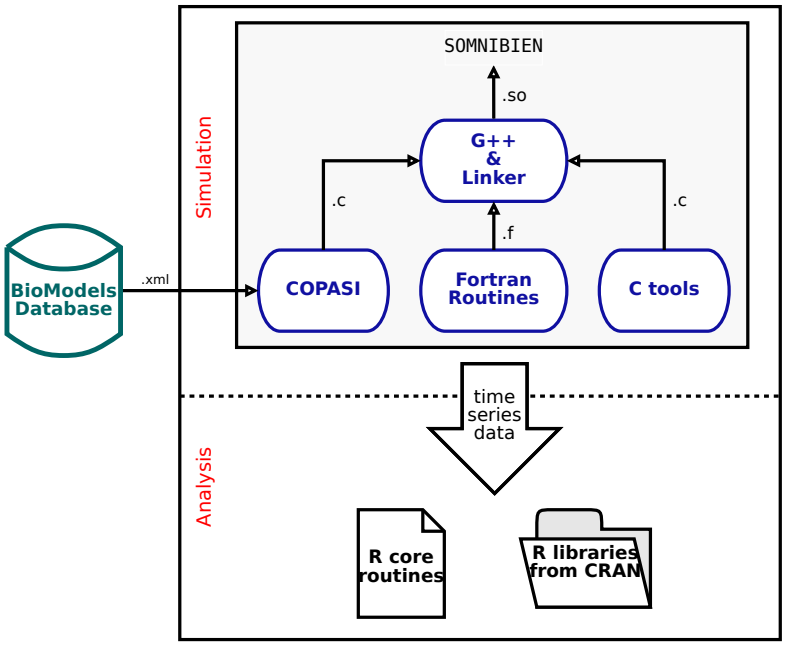

Figure 2: Basic workflow of our system. Models are downloaded as XML-files from the Internet or loaded from the file system. Various calls to external components (COPASI, g++ compiler, linker) compile SBML to C, combine this with numerical code in Fortran, and tool infrastructure in C into an object file, that can be called from within SOMNIBIEN and therefor R. This enables the user to analyze the simulated data *directly* within R using its superb analysis procedures as well as additional R-software from repositories such as CRAN [cra].

One could – in theory as it turned out – also combine existing R solvers such as deSolver [SPS10] with the R-based SBML-library SBMLR [Rad04] to achieve the same. Unfortunately, we were never able to a) integrate these components seamlessly, b) always get a working system that did not crash, and c) deal with the various SBML file format revisions and versions [Huc03, FH03b, sbm].

4 Lessons Learned

A first development approach intended to compile the Fortran and the C code containing the SDEs into shared objects during R execution. A pointer from the SDE function should then be passed to the also dynamically loaded solver in Fortran. Figure 3 illustrates the intended architecture of the package.

This leads to two problems, both caused by the fact that R treats function symbols from shared objects internally as objects. First, pointers to objects in R cannot be retrieved from

Listing 1: Typical workflow of the SOMNIBIEN package.

```
library("SOMNIBIEN")

## Downloading model from BioModels Database based on the ID (right now
    only curated Models are supported). This will  download an SBML file
    to your current working directory
getModel(42)

## Generating the shared object
PrepareODEFile("BIOMD0000000042.xml", filename="Bio042")

## Extracting the necessary information of the network from the C file.
    Note that PrepareODEFile generated a Copasi-style C file in background
InitConc <- GetCopasiInitialValues("BIOMD0000000042.c")
Parameters <- GetCopasiParameterValues("BIOMD0000000042.c")

## Starting the simulation
SimData <- StartSim(sfile="Bio042.so", parameters=Parameters,
    concentrations=InitConc, stepwidth=0.01)

## Evaluation with cross correlation function
EvalData <-  evalTimeseries(SimData, method="ccf")
```

the C/Fortran code, so passing function pointers to the SDERK-solver is not possible. Second, the Fortran interface to `R` does only supports primitives to be passed. Consequentially, `R` cannot pass functions or pointers to functions.

Due to this problem we had to develop additional C code which we name "controller". Figure 4 shows a schematic of the final architecture. The controller and the solver are compiled into a shared object during *installation* of the R package and thus only once. When the simulation is started via the `R` REPL or in `Rscript` batch mode, the wrapper compiles an SBML file containing the model into C; then the full SDE model file is compiled into a shared object that is in turn loaded *dynamically* from the controller, which passes the function pointer to the actual solver SDERK.

Since the Fortran solver requires the afore mentioned C files, we use the command line version of COPASI to convert the SBML markup into C code before running it with the `sderk90` solver. This whole process, however, is integrated into our `R` package, which will recognize if an SBML file is passed instead of a C file and automatically convert it to the required format in the background – letting the user experience a smooth integration without the need to understand the particulars "under the hood".

This involved flow of information, (function) pointers, etc. was necessary to achieve all of the stated above goals. In Listing 2 we illustrate in detail the code necessary for the pointer arithmetic and conversion to interface the components. The overall architecture fulfills all requirements from above and can be used by non-experts. However, it is also fragile against subtle changes in some of the components, in particular in the solver itself. The overall message at this point is that without loss of performance, one can integrate various tools and codes to offer an integrated, HPC-enabled simulation infrastructure.

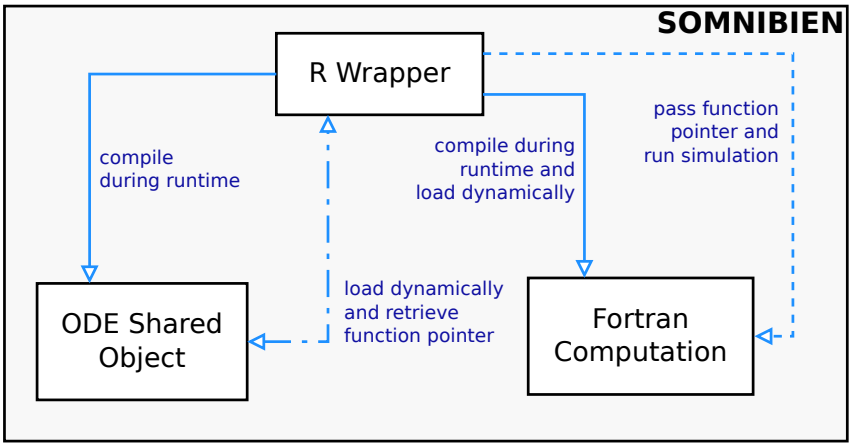

Figure 3: Initially intended package architecture. An interactive or batch-mode `R` process compiles the SDE function and the FORTRAN code during runtime and loads both. Then a pointer to the SDE function is retrieved and passed to the Fortran solver.

4.1 Parallelization

4.1.1 Parallelization of external noise

We further proceeded to improve the performance by various parallelization strategies. First, *external* noise in the form of parameter changes can easily be mapped onto SIMD systems as each simulation for a perturbed parameter set[5] is independent of any other simulation – thus, we face the rather undemanding challenge of parallelizing an embarrassingly parallel [Fos95] problem. This can be achieved within `R` itself, e.g., via the `snow` package [TRL09]. We have already discussed how the `snow` package performs very well in biophysical simulations [HWH10].

Therefore, at this point we can almost perfectly parallelize parameter scans and the analysis procedures following the myriads of simulations afterwards. This applied in particular to "grid scanning" of the parameter space as is frequently done, cmp. [XK07].

4.1.2 Parallelization of analysis steps

Analysis of trajectories and simulated data in general can sometimes be as time consuming as the simulation itself. Therefore, any efficiency improvement by, e.g., parallelization, is beneficial.

We have – as a proof of concept – implemented the computation of all cross-correlation functions (ccf) $a_{ij}(\tau) := \sum_{t=0}^{T-\tau} M_i(t) \cdot M_j(t+\tau)$ for the metabolite concentrations M_i

[5] such as the rate constants $v_i^{\downarrow}, v_i^{\uparrow}$ in Eq. 4

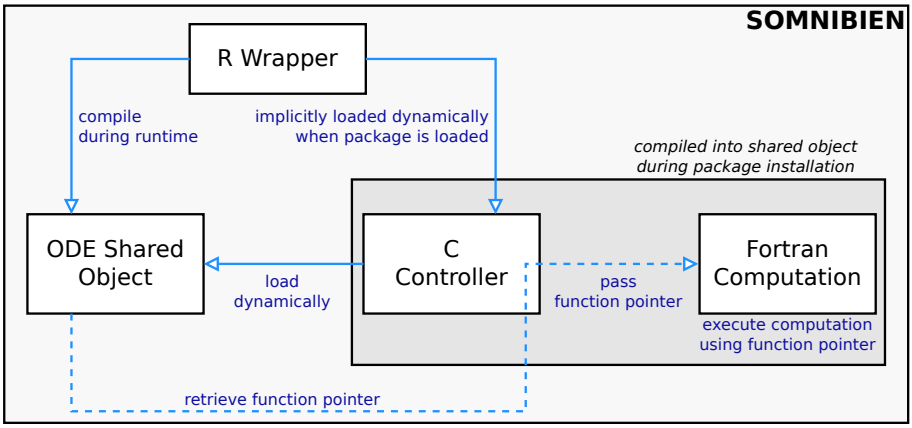

Figure 4: Overview of the implemented package architecture. The C and Fortran sources are compiled into one shared object during package installation. Each time the package is loaded in `R` using the `library` command, this shared object is loaded, too. The SDE shared object is compiled during `R` runtime. When starting a simulation the controller part of the SOMNIBIEN shared object loads the SDE shared object and retrieves a pointer to the SDE function, which is passed to and stored in the Fortran part during the simulation.

and M_j in parallel. This can be done very efficiently and easily with add-on packages in `R`, for example via the `doParallel` package. Since almost all analysis is done for all of the time series, this straight-forward approach is almost always applicable.

Note that both parallelization steps in Secs. 4.1.1 and 4.1.2 are very obviously parallel and thus – as long as there are more sub-jobs than processing elements perfectly parallelizable [MSM04].

4.1.3 Parallelization of single instances

Now, in optimization and network design one frequently deals with only one system and thus parameter set at a time – for example in Simulated Annealing in similar approaches [Ham13, Ham06], the optimization procedure would need to simulate just *one* replica of a system, propose a new one with varied parameters or graph topology, estimates its properties via simulation, and so on. Here, high-level parallelization over data-parallelism as discussed above would not be of any use. We therefore also tried to parallelize an *individual* simulation based on our architecture. To this end, we have employed the OpenMP parallelization paradigm to the evaluation of the right-hand side of the SDEs in Eq. 4 and modified the compilation routines to this effect.

A direct parallelization of the Runge-Kutta algorithm in `sderk90` is prohibitive, as the prediction and correction steps imply heavy thread creation and collapse, which caused a too large overhead during our tests to be of any use.

Our empirical results on the parallelization performance evaluation are shown for the net-

works of Tab. 1 in Figs. 5 and 6. Obviously there are only marginal, if at all efficiency improvements. Sometimes the OpenMP threading created so much overhead while the computational costs of the evaluation of the rhs was so small, that no break-even point exists. We conclude that the costs parallelization of a `single` instance is prohibitive.

Table 1: Overview of networks features: Number of metabolites, parameters and reactions

ID	Metabolites	Parameters	Reactions	ID	Metabolites	Parameters	Reactions
Mod042	15	25	25	Mod320	9	31	14
Mod160	19	47	41	Mod422	22	13	43
Mod209	6	49	12	Mod505	45	140	59

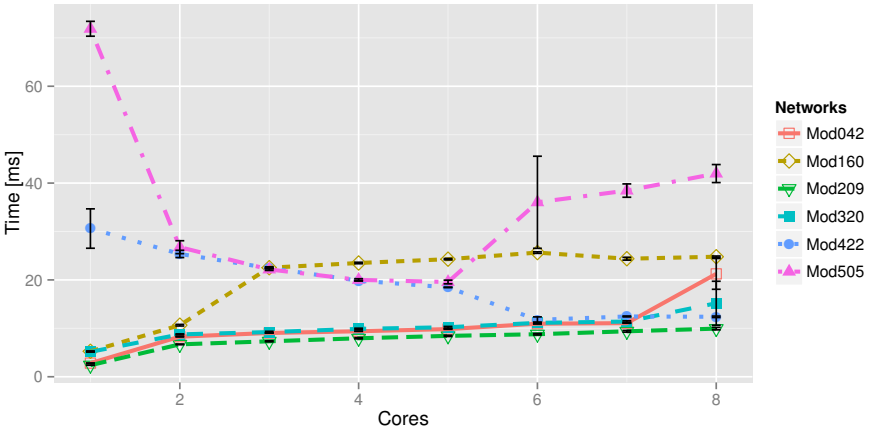

Figure 5: Execution times of an integration step averaged over the six different metabolic networks of Tab. 1 without internal noise ($\beta = 0$ in Eq. 4). Each network was simulated 100 times on the given number of cores. Error bars show the standard deviation.

5 Conclusion and Outlook

The package we presented, gives life scientists a tool to investigate the dynamics of (metabolic) networks without switching between different software systems, even allowing for large-scale computations on HPC structures. In the beginning we defined some important goals such as *interactive* and *HPC, stand-alone* usage; integration with `R` for analysis, as well as efficiency and "one-stop" usage. All these aspects are addressed by SOMNIBIEN.

Our contribution is a recipe how to deal with the particulars of `R` as a system omnipresent in the life sciences. Other approaches – such as the SimTech Workflow Management System [SKL10] – are more general and/or applicable to HPC settings but not frequently used in the life sciences where workstations and group clusters are standard.

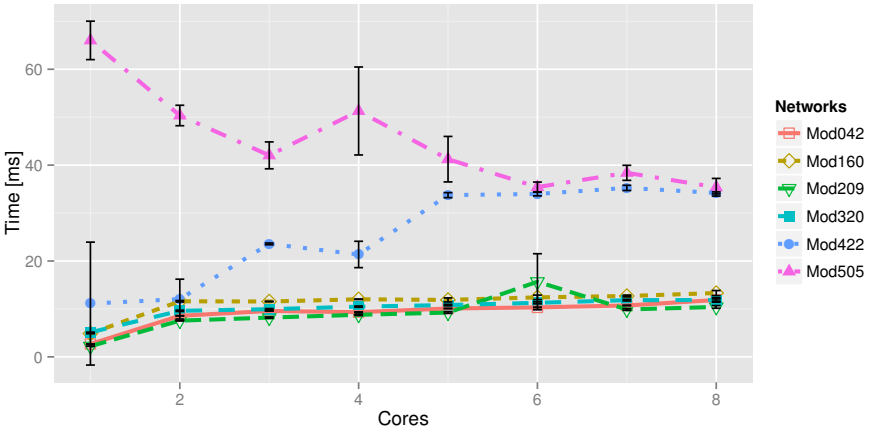

Figure 6: Same as in Fig. 5, but for a noise level of $\beta = 0.01$ in Eq. 4.

However, we had to overcome several obstacles which we encountered. Most of them are a direct results of R's layout and mechanism to handle objects and foreign-function-interfaces (FFI). We developed a mechanism via pointers in C and Fortran to integrate various tools. While the overall architecture is complex for the developer and maintainer, the user can handle the complex tasks at an abstract level. The overall method did not lead to any noticeable overhead in the computational efficiency.

Our parallelization strategies are two-fold: while the first one over independent replica parallellizes almost perfectly, our second approach to leverage OpenMP for a single instance showed problems due to high overhead costs. We hope that our short description helps other simulation and HPC researchers to integrate their systems with R, as it has become the predominant platform for end-users and analysts in recent years.

References

[BGH14] P. Boba, S. Gries, and K. Hamacher. SOMNIBIEN: Simulation Of Metabolic Networks Influenced By Internal And External Noise, 2014. *to be submitted*, available from http://www.cbs.tu-darmstadt.de/somnibien.

[cra] The Comprehensive R Archive Network. http://cran.r-project.org/.

[FH03a] A. Finney and M. Hucka. Systems biology markup language: Level 2 and beyond. Biochemical Society Transactions, 31(6):1472–1473, 2003.

[FH03b] A Finney and M Hucka. Systems biology markup language: Level 2 and beyond. Biochem Soc Trans, 31:1472–1473, 2003.

[Fos95] Ian Foster. Designing and building parallel programs : concepts and tools for parallel software engineering. Addison-Wesley, Reading, Mass, 1995.

[Fun03] A. etal Funahashi. CellDesigner: a process diagram editor for gene-regulatory and biochemical networks. Biosilico, 1(5):159–162, 2003.

[GGJ+11] E. Gehrmann, C. Gläßer, Y. Jin, B. Sendhoff, B. Drossel, and K. Hamacher. Robustness of glycolysis in yeast to internal and external noise. Phys. Rev. E, 84:021913, 2011.

[Ham06] K. Hamacher. Adaptation in stochastic tunneling global optimization of complex potential energy landscapes. Europhys. Lett., 74(6):944–950, 2006.

[Ham13] K. Hamacher. A New Hybrid Metaheuristic - Combining Stochastic Tunneling and Energy Landscape Paving. In M. Blesa, C. Blum, P. Festa, A. Roli, and M. Sampels, editors, Hybrid Metaheuristics, volume 7919 of Lecture Notes in Computer Science, pages 107–117. Springer Berlin, 2013.

[HSG+06] 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.

[Huc03] M. et al Hucka. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics, 19(4):524–531, 2003.

[HWH10] F. Hoffgaard, P. Weil, and K. Hamacher. BioPhysConnectoR: Connecting Sequence Information and Biophysical Models. BMC Bioinformatics, 11(1):199, 2010.

[KLW+09] E. Klipp, W. Liebermeister, C. Wierling, A. Kowald, H. Lehrach, and R. Herwig. Systems biology. John Wiley & Sons, 2009.

[LMBS13] Carlos F Lopez, Jeremy L Muhlich, John A Bachman, and Peter K Sorger. Programming biological models in Python using PySB. Molecular Systems Biology, 9(1), 2013.

[MSM04] T. Mattson, B. Sanders, and B. Massingill. Patterns for Parallel Programming. Addison-Wesley, 2004.

[Rad04] Tomas Radivoyevitch. A two-way interface between limited Systems Biology Markup Language and R. BMC Bioinformatics, 5(1):190, 2004.

[sbm] Systems Biology Markup Language. `http://sbml.org`.

[SKL10] M. Sonntag, D. Karastoyanova, and F. Leymann. The Missing Features of Workflow Systems for Scientific Computations. In 3rd Grid Workflow Workshop (GWW), Lecture Notes in Informatics, pages P–160, 2010.

[SPS10] Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer. Solving Differential Equations in R: Package deSolve. J. Stat. Softw., 33(9):1–25, 2010.

[Ste08] Daniel A. Steck. SDERK90, A Fortran 90 Integrator for Ito Stochastic Differential Equations, 2008. available at `http://steck.us/computer.html`.

[TRL09] Luke Tierney, A.J. Rossini, and Na Li. Snow: A Parallel Computing Framework for the R System. International Journal of Parallel Programming, 37(1):78–90, 2009.

[XK07] Z. Xie and D. Kulasiri. Modelling of circadian rhythms in Drosophila incorporating the interlocked PER/TIM and VRI/PDP1 feedback loops. J. Theo. Biol., 245(2):290 – 304, 2007.

Listing 2: Code snippet from the C controller, that mediates pointer between the Fortran solver `sderk90` and the shared object of the network's SDE system. `odefun` is the function to be integrated and `setFA` manipulates a global pointer variable. Furthermore, the code allows for a user-provided noise function in Eq. 4 via the β argument – if its name is given, then an object file is loaded and integrated into the final .so file.

```
...
// forward-define Fortran functions
void setFA(void (**)(double*, double**, double**, double*));
void setDrift(void (**)(double*, double**, double*));
void setBeta(double*);
...

// passing parameters as s-expressions
SEXP launchSim(SEXP infile, ..., SEXP trmethod)
{
...
    void * clib;
    PROTECT(infile = coerceVector(infile, STRSXP));
    // load given shared object and extract needed function symbol
    clib = dlopen(CHAR(STRING_ELT(infile, 0)), RTLD_NOW | TLD_LOCAL);
    // load dynamic library "infile"
    PROTECT(fnname = coerceVector(fnname, STRSXP));
    // allocation of 'ode_fun' using "fnname" function from "infile"
    *(void **)(&ode_fun) = dlsym(clib, CHAR(STRING_ELT(fnname, 0)));
        UNPROTECT(2);
    // pass function pointer to fortran
    setFA(&ode_fun);
    void * ex_drift;
    if(isNumeric(beta)) {
        PROTECT(beta = coerceVector(beta, REALSXP));
        // set beta value for internal noise
        setBeta(REAL(beta));
        UNPROTECT(1);
    } else if(isString(beta)) {
        // basically do the same with drift function as with ODE
        // function, if custom one is to be used.
        // function pointer declaration 'drift_fun':
        void (*drift_fn)(double*, double**, double*);
        PROTECT(beta = coerceVector(beta, STRSXP));
        PROTECT(betafname = coerceVector(betafname, STRSXP));
        // load dynamic library "beta"
        ex_drift = dlopen(CHAR(STRING_ELT(beta, 0)), RTLD_NOW | RTLD_LOCAL
            );
        // load "betafname" symbol from "beta" into 'drift_fun'
        *(void**)(&drift_fn) = dlsym(ex_drift, CHAR(STRING_ELT(betafname,
            0)));
        UNPROTECT(2);
        // set drift function pointer in Fortran
        setDrift(&drift_fn);
    } else {fprintf(stderr,"Error in beta argument"); exit(1); }
...
}
```