OneModel: an open-source SBML modeling tool focused on accessibility, simplicity and modularity ⋆

F. N. Santos-Navarro * **J. L. Navarro** * **Y. Boada** * **A. Vignoni** * **J. Picó** *

* *Synthetic Biology and Biosystems Control Lab, I.U. de Automática e Informática Industrial (ai2), Universitat Politècnica de Valencia, 46022, Camino de Vera S/N, Valencia, Spain.*
(e-mail: {fersann1,jlnavarr,yaboa,alvig2,jpico}@upv.es)

Abstract: With the advent of the Systems Biology Markup Language (SBML), a large community of SBML-compliant tools has been created. However, these tools can only be used to their full potential by expert users with advanced programming knowledge. *OneModel* is an open-source text-based tool for defining SBML models in a modular and incremental way that minimizes the user's programming knowledge requirements. It is focused on accessibility, simplicity, and modularity. *OneModel* syntax allows the user to define models based on chemical (and pseudo-chemical) reactions, differential equations, and algebraic equations. *OneModel* is written in *Python*, and it provides two interfaces: a command-line interface for expert-users, and a graphical user interface for non-expert users. Here, we show two *OneModel* syntax use case scenarios for modeling an antithetic controller and then integrating it into a host-aware model, which is freely distributed with *OneModel*.

Keywords: systems biology; synthetic biology; mathematical models; SBML; Python

1. INTRODUCTION

The Systems Biology Markup Language (SBML) is the software data format for describing models in biology (Hucka et al., 2019). With the advent of SBML, many SBML-compliant tools have been created. These tools fulfill the syntax and semantics of SBML through different approaches: text-based tools such as *Antimony* (Smith et al., 2009), *Little b* (Mallavarapu et al., 2009), *BioCRNpyler* (Poole et al., 2020); or graphical user interface based tools such as *CellDesigner* (Funahashi et al., 2008).

Models are often constructed as a monolithic set of equations, reactions, parameters, and species (Mallavarapu et al., 2009). This leads to inefficient modeling practices in which (i) new models are implemented from scratch, rather than extending previous models; (ii) models have to be validated as a whole, rather than validating the constituent parts of the model; and (iii) models tend to be large and repetitive, rather than defining and reusing modules in their implementation. *Antimony*, *BioCRNpyler* and *Little b* solve this problem by implementing different degrees of modularity. However, these tools are aimed at, or can only be used to their full potential by, expert users with advanced programming knowledge.

OneModel is an open-source text-based tool for defining and compiling SBML models in a modular way. This modularity allows incremental implementations of several simple models to obtain a more complex one. *OneModel* also minimizes the user's programming knowledge requirements. *OneModel* was designed to be easy-to-use and easy-to-incorporate into pre-existing workflows. We used well-documented *Python* libraries to avoid custom code development in its implementation. Therefore advanced programmers will be able to tweak, expand or hack *OneModel* functionality easily. The syntax of our tool implements modularity through object-oriented programming. We were inspired by the Arduino community, where a simple graphical user interface enables non-expert users to contribute their work and ideas to the community.

⋆ This work was partially funded by by Grant MINECO/AEI, EU DPI2017-82896-C2-1-R and MCIN/AEI/10.13039/501100011033 grant number PID2020-117271RB-C21. F.N. Santos-Navarro is grateful to grant PAID-01-2017 (Universitat Politècnica de València). Y. Boada thanks to Secretaría de Educación Superior, Ciencia, Tecnología e Innovación-Ecuador (Scholarship Convocatoria Abierta 2011) and Universitat Politècnica de València (Grant PAID-10-21 Acceso al Sistema Español de Ciencia e Innovación).

2. MATERIALS AND METHODS

2.1 OneModel design philosophy

OneModel was developed to meet the following design requirements in systems and synthetic biology:

- **Reactions:** to define models based on reactions with linear or rational rates (e.g. a Hill function) that depend on reactant concentrations.
- **ODE:** to define models based on ordinary differential equations (ODE).
- **DAE:** to define models based on ODE and differential-algebraic equations (DAE).
- **Modularity:** to define models incrementally and reuse specific model parts or functions.
- **Accessibility:** low entry barriers for non-expert users, and ease to integrate with other available tools.
- **Simplicity:** the tool's scope is limited to the definition and generation of SBML models; and the simplicity of the tool's internal implementation.
- **Open source:** the code is freely available to the public.

Table 1. Software available compared with the requirements. Green tick (fully met), yellow tick (partially met), and gray dash (not met).

Requirements	Antimony	Little b	BioCRNpyler	OneModel
Reactions	✓	✓	✓	✓
ODE	✓	✓	✓	✓
DAE	–	–	–	✓
Modularity	✓	✓	✓	✓
Accesibility	✓	–	✓	✓
Simplicity	✓	✓	✓	✓
Open source	✓	✓	✓	✓

Most of the available text-based tools fail to meet these requirements. Table 1 enumerates the tools which best met our design requirements.

Antimony, *Little b* and *BioCRNpyler* allow the user to define models based on reactions and ODE, but none of these tools supports algebraic equations (DAE), an inherent element of the reduced-order models generated by the quasi-steady-state (QSS) approximation. They also provide enough modularity for model definition. *Antimony* had some minor problems that limited its full potential (but they will most likely be fixed in the following versions).

About accessibility, *Antimony* is very accessible through the use of *Tellurium* (Medley et al., 2018), it defines its domain-specific language (as *OneModel* does), and the need of knowing *Python* is just for simulation and analysis of the generated models (not the definition of them). *BioCRNpyler* is very accessible but does not define a domain-specific language, and it relies on *Python* knowledge for the definition of the models. *Little b* does not meet our requirements for accessibility.

Concerning simplicity, the three tools are focused on the definition of SBML models. *Antimony* defines its custom syntax parser that is a handicap when one looks for the simplicity of the tool's internal implementation: it will make it harder to understand, extend, and maintain the tool's code by external developers. *Little b* source code was not found by the author. *BioCRNpyler* internal implementation is available and is based on Python; therefore, the simplicity requirement is satisfied.

All three tools are freely distributed, but we did not find the source code for *Little b*.

OneModel allows the user to define models with chemical or biochemical pseudo-reactions; and differential and algebraic equations. It has sufficient modularity to implement complex models efficiently. In addition, *OneModel* defines a domain-specific language (to avoid learning *Python* by the user) and incorporates two interfaces: the graphical user interface, which lowers the entry barriers for non-expert users to the minimum, and the command-line interface for expert users to integrate *OneModel* into their workflows. It is focused on definition of SBML models and it minimizes the use of custom code in its implementation. Finally, it is freely distributed and its source code can be found at `https://github.com/sb2cl/onemodel`.

2.2 *OneModel implementation*

OneModel was implemented in *Python* because it is an open-source programming language, it is easy-to-learn, and it bridges the compatibility gap with other programs. Lastly, its extensive libraries facilitated *OneModel* development.

OneModel defines a domain specific language (DSL): the *OneModel* syntax. This syntax has been implemented using *TatSu*, which allows us to create syntax parsers efficiently. This makes the *OneModel* syntax easily modifiable and adaptable. One advantage of developing a domain-specific language (instead of having implemented just a *Python* library) is that it lowers the entry barriers for the user: there is no need to learn *Python*. Examples of successful domain-specific languages are HTML (HyperText Markup Language) and CSS (Cascading Style Sheets), pseudo programming languages for non-expert users. In addition, the use of a domain-specific language allows the definition of high-level concepts (such as functions, classes, etc.) that are not currently available in SBML. *OneModel* uses libSBML, a library that simplifies reading and writing SBML files, and it is widely used in the SBML community to export models as SBML code.

Figure 1 shows the internal structure of *OneModel*. The core functionality was developed as a *Python* package. *OneModel* provides two different interfaces to simplify and abstract the use of the *Python* package: the command-line interface, and the graphical user interface. The command-line interface can be used directly by an expert user, and it has been developed with Click, a package that allows us to implement professional command-line interfaces. However, using a command-line interface is far from ideal for a non-expert user. Figure 2 shows the *OneModel* graphical user interface. It abstracts the use of the command-line interface, and it is a good interface for non-expert users. The graphical user interface was built using *PyQt5*, a *Python* package for developing graphical-user interfaces that can run in any operating system.

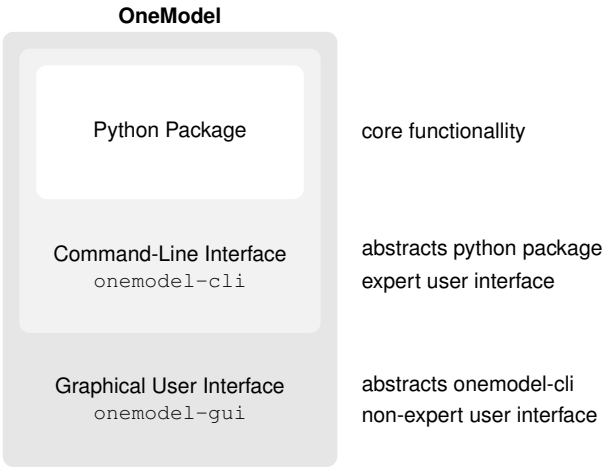

Fig. 1. Internal structure of *OneModel*. Its core is a *Python* package. The command-line interface abstracts the functionality of the python package, and the graphical user interface abstracts the command-line interface.

2.3 *OneModel syntax*

The *OneModel* syntax simplifies the definition of SBML models and extends the functionality of SBML by introducing high-

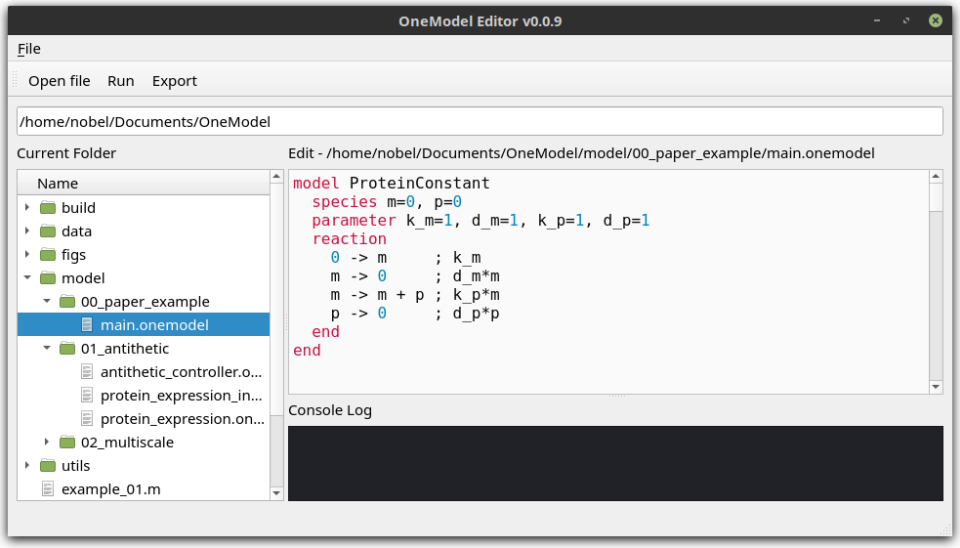

Fig. 2. *OneModel* graphical user interface running in Linux Mint 19. This graphical interface can be setup in Windows, Mac, and Linux, and it provides a simple text-editor with a syntax highlighter for *OneModel*.

```
1  import dependencies
2  model ModelName(ParentModelName)
3    define inputs
4    define species
5    define parameters
6    define reactions
7    define rules
8  end
9  standalone
10   example script on how to use the model
11 end
```

Fig. 3. Pseudo-code representing the structure of a model written with *OneModel* syntax. In line 1, the user can import previously defined models. In lines 2–8, the user can define one o more models using: inputs, species, parameters, reactions, and rules. The models might be an extension code of the ones previously defined. Finally, in lines 9–11, the user can define an example-of-use instance for the models in this file and within the standalone block.

level elements. The models can be defined using base SBML elements such as parameters, species, reactions, rules (substitution, differential or algebraic equations); or using *OneModel* high-level elements such as functions, classes, inheritance. High-level elements, which do not have an SBML representation, are converted into equivalent low-level representations when the model is exported to SBML.

Figure 3 shows the typical structure of a model developed with *OneModel*. There are three main sections: import dependencies from previously defined models (line 1), the model definition (lines 2–8), and the standalone example (lines 9–11).

The first section imports all the dependencies. Here, the user imports previously defined models to use and combine them as building blocks for developing more complex models or to extend their functionality.

The second section is the model definition. In this section, several models can be defined using: (i) inputs, (ii) species, which can be interpreted both as chemical species and state variables, (iii) parameters used by reactions and rules, (iv) biochemical reversible and irreversible reactions, and (v) rules which can be substitution, differential or algebraic equations. Models can extend the functionality of previous ones; for example the model with name "`ModelName`" will inherit all the elements (inputs, species, parameters, etc) of the parent model with name "`ParentModelName`" (line 2).

The last section is the standalone instance. Within this block, the user can define an example code that shows how to use the model (or models) defined above. It is important to note that the code inside the standalone block is not imported: the standalone code is only read when we export this model file directly. The advantages are: (i) each model file can always be exported as a standalone model allowing us to test each model individually for coherence, and (ii) the user always has an example of how to use the model.

2.4 Subpackage SBML2dae

We have also created SBML2dae, a *OneModel* subpackage which provides tools to generate SBML exporters to other programming languages for simulation or analysis. By default, SBML2dae only allows exporting SBML to Matlab. The differences with other Matlab converters are (i) SBML2dae allows the simulation of algebraic loops (an indispensable element for the simulation of reduced order models, using the QSS aproximation), (ii) it generates Matlab code using classes which greatly facilitates the integration of the models with the rest of Matlab tools and (iii) SBML2dae is easily modifiable to change the way of exporting the models.

3. RESULTS

To simplify and streamline the modeling tasks, *OneModel* allows us to create incremental implementations of a model by using smaller models previously defined. In the following sections we present two case scenarios to show how to work with *OneModel* and how these ideas can be carried out with its syntax is an easy way.

These case scenarios demonstrate how to use *OneModel* software. A more complete and commented version of these examples can be found on *Github* (`https://github.com/sb2cl/Dycops21_OneModel`). Further details can be found in the *OneModel* syntax documentation (`https://onemodel.readthedocs.io/en/latest/`).

3.1 OneModel workflow

This section describes the *OneModel* workflow (Figure 4) to help understand its use and usefulness.

The first step is to write the model, as a plain text file with ".one" or ".onemodel" as the extension, using *OneModel* syntax. The user can use either the *OneModel*'s editor (available in the graphical user interface) or his own text editor. The second step is to export this model as an SBML file (both the GUI and CLI can do this task). Then, the SBML file can be fed into any available SBML-compliant tools to perform the computational simulations, analysis, etc. Finally, once the model has been validated, we can repeat this loop, generating a new model that imports the code and functionality of the previously defined models. In the following sections, *SBML2dae* is used (already included in the *OneModel* GUI) to generate a *Matlab* implementation of the generated SBML model. But we could also have used *SBML2Modelica* to generate a *Modelica* implementation of the model instead.

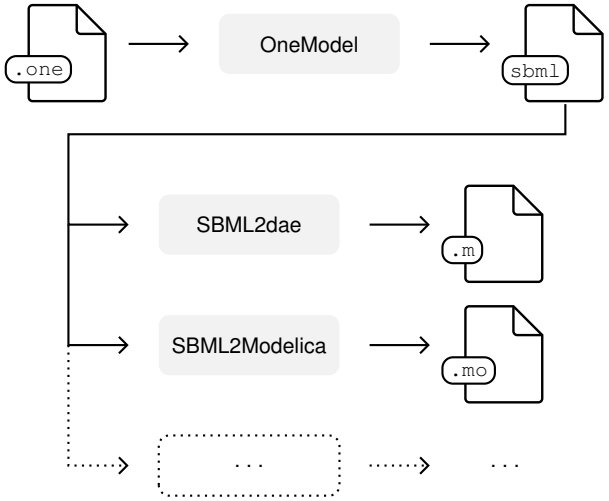

Fig. 4. *OneModel* workflow. The user writes a model using *OneModel* syntax (".one"). Then, the model is exported into SBML using *OneModel*. Finally, the SBML-compliant tools can be used as (i) *SBML2dae* generates a *Matlab* implementation of the model (".m"), or (ii) *SBML2Modelica* (Maggioli et al., 2020) generates a *Modelica* representation (".mo").

3.2 Case 1: antithetic controller implementation

The antithetic controller is a synthetic gene system to robustly control the expression of a protein of interest (Aoki et al., 2019). This circuit is implemented by three genes coding three different proteins: sigma `z1`, anti-sigma `z2`, and the protein of interest `x`. Normally, `z1` is constitutively expressed and induces the production of `x`. In turn, `x` activates the expression of `z2`. Finally, `z1` and `z2` annihilate each other in a sequestration reaction, closing the loop.

First, we have to define the biochemical reactions that take place in this gene circuit. Protein `z1` constitutive expression entails: transcription and translation, and degradation of both mRNA and protein:

$$\begin{aligned} \emptyset &\xrightarrow{k_m} \mathrm{m_{z1}} \\ \mathrm{m_{z1}} &\xrightarrow{d_m} \emptyset \\ \mathrm{m_{z1}} &\xrightarrow{k_p} \mathrm{m_{z1}} + \mathrm{z_1} \\ \mathrm{z_1} &\xrightarrow{d_p} \emptyset \end{aligned} \tag{1}$$

where m_{z1} is the mRNA concentration; k_m and k_p are the rate parameters related to transcription and translation, respectively; and d_m and d_p are the degradation rates of the mRNA and the protein `z1`.

To make a modular implementation of the antithetic controller, the model of a generic protein with constitutive expression will be implemented first. Figure 5 depicts the *ProteinConstitutive* model implemented in *OneModel* using the reactions of `z1` in (1) as a guide, but implemented as protein `p` and mRNA `m`. In line 2, we defined the two species m and `p`, and then we set their initial conditions as $m(0) = p(0) = 0$. In line 3, we defined the values of the parameter (in this case the reaction rates values). Next, using the reactions in (1) for z_1 as a guide, we implement the reactions required to produce protein `p` are in lines 4 to 9. Lastly, we defined the standalone example for this model as a simple example of a protein constitutively expressed. Recall that, the code inside the standalone block wont be imported when we import this model into other one.

protein_constitutive.one

```
1  model ProteinConstitutive
2    species m=0, p=0
3    parameter k_m=1, d_m=1, k_p=1, d_p=1
4    reaction
5      0 -> m      ; k_m
6      m -> 0      ; d_m*m
7      m -> m + p  ; k_p*m
8      p -> 0      ; d_p*p
9    end
10 end
11 standalone
12   A = ProteinConstitutive()
13 end
```

Fig. 5. *OneModel* implementation of the biochemical reactions set 1. *ProteinConstitutive* defines transcription and translation (basic mechanism) of a protein constitutively expressed.

Figure 6 illustrates the *ProteinInduced* model, which extends *ProteinConstitutive* model so we can obtain a new model for protein expression regulated by a transcription factor. First, we imported the code from "protein_constitutive.one" into this model (line 1). Then, we defined *ProteinInduced* model as an extension of *ProteinConstitutive* by placing its name inside the parentheses in the definition (line 2). We defined the transcription factor (`TF`) as an input to this model in line 3. When we use this model, we will have to satisfy the input value, with a species, a parameter or an equation from another model. Next, we set the parameters for a linear induction (line 4). We have to override the parameter `k_m`, which is the transcription rate, to be a species (note that a species refers both to chemical

species and state variables) which can change over time (line 5). Then, We assign its value as a substitution equation (line 7) which proportionally depends on the value of `TF`. Finally, we set up a standalone example where we had the protein expressed constitutively (A) using the code in Fig. 5, and we induce the expression of a protein `B` by the concentration of `A`.

protein_induced.one

```
1  import "protein_constitutive.one"
2  model ProteinInduced(ProteinConstitutive)
3    input TF
4    parameter h=1, k_m_max=1
5    species k_m=0
6    rule
7      k_m := k_m_max*TF/h
8    end
9  end
10 standalone
11   A = ProteinConstitutive()
12   B = ProteinInduced()
13   rule
14     B.TF := A.p
15   end
16 end
```

Fig. 6. *ProteinInduced* model represents the induction of protein production by extending the code of *ProteinConstitutive*.

Figure 7 shows the *AntitheticController* model using the models defined above. First, we import the previous models into this new model (lines 1–2). We define the three proteins which make the motif: we use *ProteinConstitutive* for protein `z1` and *ProteinInduced* for proteins `z2` and `x` (lines 4–6). We define the annihilation rate `gamma` (line 7) and we add the annihilation reaction to the model (line 9). Then, We set `z1` as the transcription factor of protein `x`. In turn, `x` will be the transcription factor of protein `z2` (lines 12–13). Finally, we set the standalone example as just the AntitheticController.

antithetic_controller.one

```
1  import "protein_constitutive.one"
2  import "protein_induced.one"
3  model AntitheticController
4    z1 = ProteinConstitutive()
5    z2 = ProteinInduced()
6    x  = ProteinInduced()
7    parameter gamma=1
8    reaction
9      z1.p + z2.p -> 0 ; gamma*z1.p*z2.p
10   end
11   rule
12     x.TF  := z1.p
13     z2.TF := x.p
14   end
15 end
16 standalone
17     ac = AntitheticController()
18 end
```

Fig. 7. Implementation of the antithetic controller using both previously define models *ProteinConstitutive* and *ProteinInduced*.

3.3 Case 2: host-aware antithetic controller model

The first approach to model a synthetic gene circuit is done by neglecting the interactions between the host and the gene circuit. However, there is an increasing need to include host dynamics to improve the model prediction capabilities. These host-aware dynamic models are complex and not easy to implement since they may contain several states (Santos-Navarro et al., 2021).

Figure 8 depicts the *WildType* model that represents the host-aware model freely distributed with *OneModel*. Lines 1–4 show an incomplete representation of it just for example purposes. This model implements the equations from (Santos-Navarro et al., 2021) and takes into account the host dynamics, the competition for cell resources in protein expression and its effect on cell growth. Here, the only requirement is to satisfy its input *WSum* (line 2) which is a value that keeps track of the burden introduced by the expression of exogenous proteins like the ones for the antithetic controller. *WildType_ProteinConstitutive* is a model provided also by *WildType*, which defines the base protein expression mechanism. The user may use this model as a building block for its own circuits (similarly to Section 3.2). There were no inputs in the original *ProteinConstitutive*. However *WildType_ProteinConstitutive* is a more complex model which has inputs to calculate protein expression as function of the effective translation rate of ribosomes `nu_t`, and the cell growth rate `mu`.

wild_type.one

```
1 model WildType
2   input WSum
3   ...
4 end
5 model WildType_ProteinConstitutive
6   input nu_t, mu, ...
7   species W
8   ...
9 end
```

Fig. 8. *WildType* is the host-aware model freely distributed with *OneModel*, and *WildType_ProteinConstitutive* is the base protein expression mechanism (incomplete representation of these models only for example purposes).

Figure 9 lists the code for the *Wildtype_AntitheticController* model of the antithetic controller above, but using *WildType_ProteinConstitutive* as the base expression mechanism for proteins. In lines 7–8, we instantiate an object of *WildType* and *AntitheticController* models. Finally, we satisfy the input of *WildType* in line 10, and the inputs `nu_t` and `mu` of the proteins in lines 11–13.

Figure 10 shows two simulations performed with SBML2dae: the antithetical controller connected to the host-aware model, and the antithetical controller not considering the burden introduced to the host. Notice how the steady state and settling time of the protein `x` changes due to the burden introduced by the antithetic controller to the host cell.

4. CONCLUSION

We developed *OneModel*, a new SBML-compliant tool for defining models focused on user accessibility, simplicity, and modularity. Instead of developing monolithic files that contain all the equations and model parameters values, *OneModel* syntax allows the user to add new models to connect with the old ones—splitting models into modules and re-programming them by making small changes that fulfill the new requirements

wild_type_antithetic_controller.one

```
import "wild_type.one"
model WildType_AntitheticController
  z1 = WildType_ProteinConstitutive()
  ...
end
standalone
  cell = WildType()
  ac   = WildType_AntitheticController()
  rule
    cell.WSum  := ac.z1.W + ac.z2.W + ac.x.W
    ac.z1.nu_t := cell.nu_t
    ac.z1.mu   := cell.mu
    ...
  end
end
```

Fig. 9. Antithetic controller model together with the host-aware model in the same code implementation.

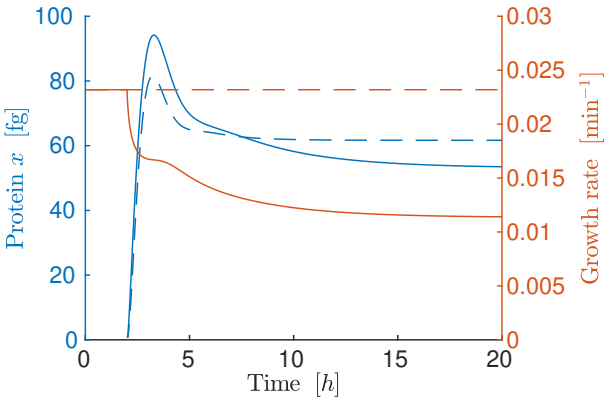

Fig. 10. Simulation of the antithetic controller with (solid lines) and without (dashed lines) the host dynamics. The protein level (solid blue) and the growth rate (solid red) decrease because the host allocates its resources for the cell fitness.

but always have the option to go back—. *OneModel* reduces the modeling efforts by increasing modularity. The user can develop and test each module of a model separately, avoiding starting from scratch every time the user implements a new model.

OneModel was tested in two case scenarios. The case scenarios showed the benefits of modular incremental implementations, and how it would be easy for a non-expert user to take advantage of previously defined models (e.g., the host-aware model).

The original motivation for building *OneModel* was that several biological processes could only be modeled with differential-algebraic equations (DAE). Our host-aware model includes this type of equations to consider the competition for shared cellular resources. Currently, *Antimony* does not support DAE. As a first iteration, we tried to add the algebraic loops to the *Antimony* source code. However, it uses a custom parser for its syntax, which restricted us modifying it. On the contrary, *OneModel* uses a well-documented parser (*TatSu*) which lowers the entry barriers for modifying *OneModel*. This way, the community will easily extend its original functionality.

OneModel requires *Python* 3.8 and is installed as a package with PyPI. Although the installation process is simple, it can be challenging for non-expert users. An executable version and a web interface will avoid this step in the future. *OneModel* syntax is Turing incomplete on purpose because that could divert its purpose from being just a tool to define SBML models (this case is similar to HTML or CSS), and this would compromise simplicity and accessibility. Error feedback is often one of the weaknesses of domain-specific languages and is key to ensuring accessibility. Currently, *OneModel* provides simple error feedback, but our goal is to improve it.

Antimony, *Little b*, *BioCRNpyler*, *OneModel*, and many other tools not listed here, paved the way to define more complex and larger models efficiently. However, these models are difficult to debug, test, and maintain. Researchers have performed these tasks manually, but this is inefficient for models of this size or sometimes even impossible. There is an increasing need for tools to automatically debug and test SBML models. Model development is a crucial task for researchers. *OneModel* allows you to simplify and streamline this task.

REFERENCES

Aoki, S.K., Lillacci, G., Gupta, A., Baumschlager, A., Schweingruber, D., and Khammash, M. (2019). A universal biomolecular integral feedback controller for robust perfect adaptation. *Nature*, 570(7762), 533–537. doi:10.1038/s41586-019-1321-1.

Funahashi, A., Matsuoka, Y., Jouraku, A., Morohashi, M., Kikuchi, N., and Kitano, H. (2008). CellDesigner 3.5: A versatile modeling tool for biochemical networks. *Proceedings of the IEEE*, 96(8), 1254–1265. doi:10.1109/JPROC.2008.925458.

Hucka, M., Bergmann, F.T., Chaouiya, C., Dräger, A., Hoops, S., Keating, S.M., König, M., Novère, N.L., Myers, C.J., Olivier, B.G., Sahle, S., Schaff, J.C., Sheriff, R., Smith, L.P., Waltemath, D., Wilkinson, D.J., and Zhang, F. (2019). The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 2 Core Release 2. *Journal of integrative bioinformatics*, 16(2). doi:10.1515/jib-2019-0021.

Maggioli, F., Mancini, T., and Tronci, E. (2020). SBML2Modelica: Integrating biochemical models within open-standard simulation ecosystems. *Bioinformatics*, 36(7), 2165–2172. doi:10.1093/bioinformatics/btz860.

Mallavarapu, A., Thomson, M., Ullian, B., and Gunawardena, J. (2009). Programming with models: Modularity and abstraction provide powerful capabilities for systems biology. *Journal of the Royal Society Interface*, 6(32), 257–270. doi:10.1098/rsif.2008.0205.

Medley, J.K., Choi, K., König, M., Smith, L., Gu, S., Hellerstein, J., Sealfon, S.C., and Sauro, H.M. (2018). Tellurium notebooks—An environment for reproducible dynamical modeling in systems biology. *PLoS Computational Biology*, 14(6), 1–24. doi:10.1371/journal.pcbi.1006220.

Poole, W., Pandey, A., Shur, A., Tuza, Z.A., and Murray, R.M. (2020). BioCRNpyler: Compiling Chemical Reaction Networks from Biomolecular Parts in Diverse Contexts. *bioRxiv*, 2020.08.02.233478.

Santos-Navarro, F.N., Vignoni, A., Boada, Y., and Picó, J. (2021). RBS and Promoter Strengths Determine the Cell-Growth-Dependent Protein Mass Fractions and Their Optimal Synthesis Rates. doi:10.1021/acssynbio.1c00131.

Smith, L.P., Bergmann, F.T., Chandran, D., and Sauro, H.M. (2009). Antimony: A modular model definition language. *Bioinformatics*, 25(18), 2452–2454. doi:10.1093/bioinformatics/btp401.