UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Brenno Lemos Melquiades dos Santos

Desenvolvimento de um software científico para a modelagem e simulação computacional

São João del-Rei

2024

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Brenno Lemos Melquiades dos Santos

Desenvolvimento de um software científico para a modelagem e simulação computacional

Dissertação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação do Programa de Pós Graduação em Ciência da Computação (PPGCC) da UFSJ.

Orientador: Alexandre Bittencourt Pigozzo

Universidade Federal de São João del-Rei — UFSJ

Pós-Graduação em Ciência da Computação

São João del-Rei

2024

Ficha catalográfica elaborada pela Divisão de Biblioteca (DIBIB)
e Núcleo de Tecnologia da Informação (NTINF) da UFSJ,
com os dados fornecidos pelo(a) autor(a)

L557d

Lemos, Brenno.
Desenvolvimento de um software científico para a modelagem e simulação computacional / Brenno Lemos ; orientador Alexandre Bittencourt Pigozzo. -- São João del-Rei, 2024.
58 p.

Dissertação (Mestrado - Ciência da Computação) -- Universidade Federal de São João del-Rei, 2024.

1. Software. 2. Modelagem Computacional. 3. Simulação Computacional. 4. Ensino-aprendizagem. 5. EDO. I. Bittencourt Pigozzo, Alexandre, orient. II. Título.

Brenno Lemos Melquiades dos Santos

Desenvolvimento de um software científico para a modelagem e simulação computacional

Dissertação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação do Programa de Pós Graduação em Ciência da Computação (PPGCC) da UFSJ.

São João del-Rei, 16 de abril de 2024

Alexandre Bittencourt Pigozzo
Orientador

Prof. Rafael Sachetto Oliveira
Convidado 1

Prof. Marcelo Lobosco
Convidado 2

Profa. Bárbara de Melo Quintela
Convidado 3

São João del-Rei
2024

Agradecimentos

Gostaria de agradecer e dedicar este trabalho:

Aos meus pais, pelas oportunidades que me deram, pelo apoio nas escolhas que fiz para chegar aqui, pelo companheirismo e pelo amor incondicional que me proporcionaram.

Ao meu orientador, Alexandre Pigozzo, que me introduziu ao meio científico, ao tema deste trabalho e me auxiliou não somente neste trabalho, como contribuidor e orientador, mas na minha trajetória nesta universidade.

Aos servidores do DCOMP-UFSJ, que mantém um alto nível de educação e demonstram respeito e empatia pelos docentes do curso. Em especial, um agradecimento aos servidores(as) Carolina Xavier, Leonardo Rocha, Marcos Laia, Michelli Loureiro e Douglas Dinalli.

Aos meus amigos de Guarapari, que sempre estiveram ao meu lado.

À minha namorada, Keina Takashiba, minha melhor amiga, que me ajuda a me organizar e me apoia em tudo que for.

Aos amigos que fiz aqui, na UFSJ, Arthur Gabriel Matos, Ana Clara Medina, Lucas Grandolpho e Wesley Guimarães, meus grandes companheiros de trabalhos.

À FAPEMIG, que apoiou este e os projetos anteriores.

Aos meus colegas e contribuidores do projeto, Diego Augusto Silva Castro, Sávio Francisco Cirino da Paz e Davi Jannotti Coelho Pinheiro.

À todos aqueles que apoiam, defendem e lutam por uma educação gratuita e de qualidade para todos.

Resumo

Palavras-chaves: modelagem, simulação, modelos computacionais, software *open source*, interface gráfica do usuário.

As áreas de Modelagem Matemática e Computacional têm se tornado cada vez mais importantes no mundo atual, no qual estudos científicos devem trazer resultados cada vez mais rápidos. Os modelos matemáticos e computacionais surgem como ferramentas poderosas no estudo e compreensão de sistemas complexos. Os modelos comumente requerem conhecimento matemático para serem criados, o que resulta em uma grande barreira de entrada para cientistas sem formação relacionada diretamente com a matemática, como biólogos, por exemplo, e estudantes iniciando carreiras acadêmicas. Apesar de existirem softwares que auxiliam o desenvolvimento de modelos computacionais, estes frequentemente apresentam interfaces muito complexas na busca para se tornarem genéricos o suficiente e fornecerem muitos recursos. Neste trabalho, é apresentado o software que foi desenvolvido para construir e simular modelos de Equações Diferenciais Ordinárias. O objetivo foi desenvolver um software simples, fácil de usar e de estender com novas funcionalidades. Através da interface gráfica, o usuário pode construir um modelo utilizando um editor baseado em nós, realizar simulações e gerar o código que implementa o modelo. O software poderá ser utilizado na pesquisa e no processo ensino-aprendizagem de modelagem computacional. O potencial do software é demonstrado através de alguns exemplos práticos.

Abstract

Key-words: modelling, simulation, automata, computational models, mathematical models, graphical interface software.

The areas of Mathematical and Computational Modeling have become increasingly important in today's world, in which scientific studies must bring increasingly faster results. Mathematical and computational models emerge as powerful tools in the study and understanding of complex systems. Models commonly require mathematical knowledge to be created, which results in a high entry barrier for scientists without mathematical training, such as biologists, for example, and students starting academic careers. Although there is software that helps the development of computational models, these often present very complex interfaces in order to become generic enough and provide many resources. In this work, it is presented a software that was developed to build and simulate models of Ordinary Differential Equations. The objective was to develop a software that is simple, easy to use and extend with new features. Through the graphical user interface, the user can build a model using a node-based editor, perform simulations and generate the code that implements the model. The software can be used in research and in the teaching-learning process of computational modeling. The potential of the software is demonstrated through some practical examples.

Lista de ilustrações

Figura 1 – Exemplo de resultado do modelo Predador-Presa clássico. 15
Figura 2 – Visão geral do ciclo da modelagem. 20
Figura 3 – Exemplo do modelo Predador-Presa (Equação 2.1) construído no *software*. 28
Figura 4 – Plotagem dos resultados da simulação do modelo Predador-Presa. . . . 29
Figura 5 – Fluxograma da experiência de usuário esperada para o software. 29
Figura 6 – Visualização dos nós Termo, Expressão e *Assigner* (ou Equação) no editor baseado em nós do software. Um exemplo de uso desses nós foi apresentado na Tabela 2. 31
Figura 7 – Árvores de expressões para cada nó de expressão no modelo predador-presa, apresentado na Figura 3. 32
Figura 8 – Os nós definidos na listagem 4.3 sendo utilizados na interface. 36
Figura 9 – Diagrama de classes do módulo da aplicação. 37
Figura 10 – Diagrama de classes do módulo de nós. 38
Figura 11 – Diagrama de classes do módulo de extensões. 38
Figura 12 – Diagrama de classes do módulo da árvor de expressões. 39
Figura 13 – Diagrama de classes do módulo de mensagens. 39
Figura 14 – Diagrama de sequência representando o processo de exportação de código, simulação interativa e exportação de resultados. 41
Figura 15 – Página principal do website do software Krita, apresentando: (1) download de executável portátil para Linux, (2) downloads de executável portátil e instalador para Windows e (3) download de instalador para Mac. 42
Figura 16 – Estrutura de diretórios da AppImage do software. 43
Figura 17 – As opções de criação de nós. Destacado como '1' está o menu lateral com as opções de nós e o campo de nome. Destacado como '2' está o menu de contexto acessível com o clique do botão direito do mouse. . . 44
Figura 18 – Os nós termos criados. Note que a constante "one" (em inglês "um") tem seu valor definido como 1. Os outros termos possuem os valores $F = 0$, $p = 0.03$ e $q = 0.38$. 45
Figura 19 – As subexpressões e termos usados para construir a Equação 4.1. Esta desconstrução é útil para a representação no software. 46
Figura 20 – As 4 expressões construídas no software: (1) qF, (2) $1 - F$, (3) $p + qF$ e (4) $(p + qF) * (1 - F)$. 46

Figura 21 – A Equação $\frac{dF}{dt} = \ldots$ descrita como um nó. Nesta etapa, é necessário selecionar um termo como variável, clicando no botão “choose” (em inglês, “escolher”). 47
Figura 22 – O modelo final construído para a Equação 4.1. Destacado como ‘1’ estão os itens que controlam os tempos das simulações, que foram alterados para 0.0 (tempo inicial), 0.1 (tempo delta) e 35.0 (tempo final). Descatado como ‘2’ está o botão para realizar a execução e simulação do programa. 48
Figura 23 – O resultado da simulação para os parâmetros $F_0 = 0$, $p = 0.03$, $q = 0.38$. 48

Figura 24 – Modelo predador-presa construído no software. 49
Figura 25 – Exemplo de resultado da simulação do modelo predador-presa. 50
Figura 26 – Modelo SIRS construído no software. 51
Figura 27 – Exemplo de resultado da simulação do modelo SIRS. 52
Figura 28 – Modelo da reação enzimática construído no software. 53
Figura 29 – Exemplo de resultado do modelo da reação enzimática. 54
Figura 30 – Modelo da infecção viral construído no software. 55
Figura 31 – Exemplo de resultado do modelo da infecção viral. 56
Figura 32 – Exemplo de resultado do modelo da infecção viral mostrando a variável E. 56

Lista de abreviaturas e siglas

EDO	Equação Diferencial Ordinária
EDP	Equação Diferencial Parcial
RI	Representação Intermediária
GUI	*Graphical User Interface*, Interface Gráfica de Usuário

Sumário

1 Introdução 12

2 Referencial Teórico 14

2.1 Equações Diferenciais Ordinárias 14

2.2 Programação Visual 15

2.2.1 Editores baseados em nós 16

2.3 Geração de código 17

2.3.1 Representação intermediária 17

2.3.2 Sistemas baseados em *templates* 19

2.4 Ensino-aprendizagem de Modelagem Computacional 19

3 Trabalhos relacionados 23

3.1 Snoopy 25

3.2 Insight maker 26

3.3 VCell 26

3.4 EpiFire 27

3.5 Cytoscape 27

4 Software para modelagem e simulação computacional 28

4.1 Uma representação visual para EDOs 30

4.1.1 Representação de expressões na interface 31

4.1.2 Uma representação intermediária humanamente legível 32

4.2 Geração de Código e Simulação interativa 34

4.3 Extensibilidade 35

4.3.1 Carregamento de extensões 36

4.4 Implementação e Tecnologias utilizadas 36

4.4.1 O módulo da GUI 36

4.4.2 A biblioteca de RI 40

4.5 Distribuição do software 40

4.5.1 Empacotamento de AppImages para sistemas Linux 42

4.5.2 Empacotamento ZIP para sistema Windows 43

4.6 Utilizando o software 43

5 Aplicação do software na modelagem computacional 49

5.1 Predador-presa 49

5.2 Transmissão de doenças 49

5.3 Reação enzimática 50
5.4 Resposta imune a uma infecção viral 52

6 Conclusão e trabalhos futuros 57

Referências 59

Apêndices 62

.1 Código de Python gerado para o modelo Predador-Presa 63
.2 RI do modelo Predador-Presa 66

1 Introdução

Modelos matemáticos e computacionais estão se tornando cada vez mais relevantes no mundo atual. Diversas áreas do conhecimento se beneficiam da simulação computacional e é através das simulações computacionais que fenômenos em várias áreas estão sendo compreendidos em diferentes escalas espaciais e temporais (MARTINS; FERREIRA; VILELA, 2010).

As simulações computacionais podem auxiliar experimentos *in vitro* e *in vivo* na explicação de diversos fenômenos. As simulações permitem testar vários cenários diferentes, o que poderia ser muito custoso ou até inviável de ser testado *in vitro* e *in vivo*. Além disso, elas podem fornecer previsões testando hipóteses que ainda não foram investigadas (WALPOLE; PAPIN; PEIRCE, 2013).

O desenvolvimento de modelos computacionais requer um conjunto de etapas importantes que começam com a definição de seu objetivo e terminam com sua validação, passando por múltiplas iterações e melhorias. Dentre estas etapas, uma das mais desafiadoras é a implementação do modelo. Essa etapa requer o conhecimento de métodos numéricos, programação, estruturas de dados, bibliotecas, entre outros recursos. Um erro na implementação pode comprometer todo o trabalho. Nesse contexto, foi desenvolvido um software para auxiliar as etapas de implementação e simulação de modelos de Equações Diferenciais Ordinárias (EDOs).

O presente trabalho pretende responder à seguinte questão científica:

- É possível que ferramentas de software automatizem etapas do processo de modelagem computacional?

Pretende-se investigar a resposta da questão anterior através da apresentação de um software que foi desenvolvido para facilitar as etapas de construção, implementação e simulação de um modelo de EDOs.

O principal objetivo deste trabalho foi desenvolver um software para facilitar a criação e simulação de modelos computacionais, diminuindo a barreira de entrada na área de Modelagem Computacional e permitindo que menos tempo seja investido na implementação do modelo. A partir do "desenho" de um modelo, o software é capaz de gerar o código que implementa o modelo, simulá-lo e até exportar gráficos com os resultados.

Entre as contribuições deste trabalho, destacam-se as seguintes:

- A criação de uma representação gráfica intuitiva para modelos matemáticos baseados

em equações diferenciais;

- O desenvolvimento de um gerador de código baseado em *templates* para gerar os códigos que implementam os modelos;
- A construção de modelos de Equações Diferenciais Ordinárias (EDOs) sem a necessidade de iniciar a implementação computacional dos modelos do zero.

Uma das aplicações do software que foi desenvolvido é no processo de ensino-aprendizagem de modelagem computacional. Os usuários poderão aprender sobre modelagem computacional por meio de um processo de aprendizado ativo com a construção e simulação dos modelos na prática. Na Seção 2.4, será discutido como o software pode ser utilizado junto com uma metodologia ativa de ensino como uma possível abordagem de ensino-aprendizagem de modelagem computacional.

O software desenvolvido chama-se ODE-Designer.

O restante do texto deste trabalho está organizado da seguinte forma: no Capítulo 2, são apresentados alguns conceitos teóricos para um melhor entendimento do trabalho. No Capítulo 3, são descritos os trabalhos relacionados. O software desenvolvido é descrito no Capítulo 4. No Capítulo 5, são apresentadas algumas aplicações do software na modelagem e, por fim, a conclusão e os trabalhos futuros são apresentados no Capítulo 6.

2 Referencial Teórico

Neste Capítulo, são apresentados os principais conceitos que formam a base teórica do trabalho.

2.1 Equações Diferenciais Ordinárias

Uma Equação Diferencial Ordinária (EDO) é um tipo de modelo matemático que descreve como as populações modeladas evoluem ao longo do tempo. Os modelos de EDOs são bastante difundidos e utilizados em diversas áreas como, por exemplo, na medicina e na neurociência (ADOMIAN, 1995), no estudo do Câncer (SPENCER et al., 2004; TALKINGTON; DANTOIN; DURRETT, 2018), no estudo da resposta imune à infecções virais (REIS et al., 2021), no estudo da eletrofisiologia cardíaca (VIGMOND et al., 2008; BUCELLI et al., 2022), entre outros exemplos.

O modelo predador-presa, também conhecido como Lotka-Volterra, é um exemplo de modelo clássico da literatura que modela a dinâmica da interação entre uma presa e um predador. O modelo é descrito pelas seguintes EDOs:

$$\begin{aligned} \frac{dH}{dt} &= r.H - a.H.P \\ \frac{dP}{dt} &= b.H.P - m.P \end{aligned} \tag{2.1}$$

As populações do modelo e as taxas são:

H	Presa
P	Predador
r	Taxa de reprodução da presa
m	Taxa de mortalidade dos predadores
a	Taxa de predação
b	Taxa de reprodução dos predadores

.

Neste modelo, temos os seguintes processos sendo modelados:

- Reprodução das presas (termo $r.H$);
- Predação (termo $a.H.P$);
- Reprodução dos predadores (termo $b.H.P$);
- Morte dos predadores (termo $m.P$).

Termos de replicação, predação e morte como os vistos neste modelo são muitos comuns e são empregados na maioria dos modelos de EDOs. Esses termos são construídos com base no princípio da Lei de Ação de Massas (*Mass Action Law*) que diz que "O número de interações entre duas partículas depende da concentração de ambas", isto é, esse princípio nos diz que quanto maior é a concentração de duas substâncias maior é a chance delas interagirem, partindo da suposição de que as substâncias em questão possuem a capacidade e "afinidade" para interagir.

A Lei de Ação de Massas pode ser aplicada, a princípio, a qualquer sistema de qualquer área onde as seguintes hipóteses são consideradas:

- O sistema é bem homogêneo (*well-mixed system*);
- As partículas/substâncias se movimentam de forma aleatória;
- Não é considerada nenhuma estrutura espacial no modelo.

Na Figura 1, é dado um exemplo de resultado obtido com a simulação do modelo predador-presa.

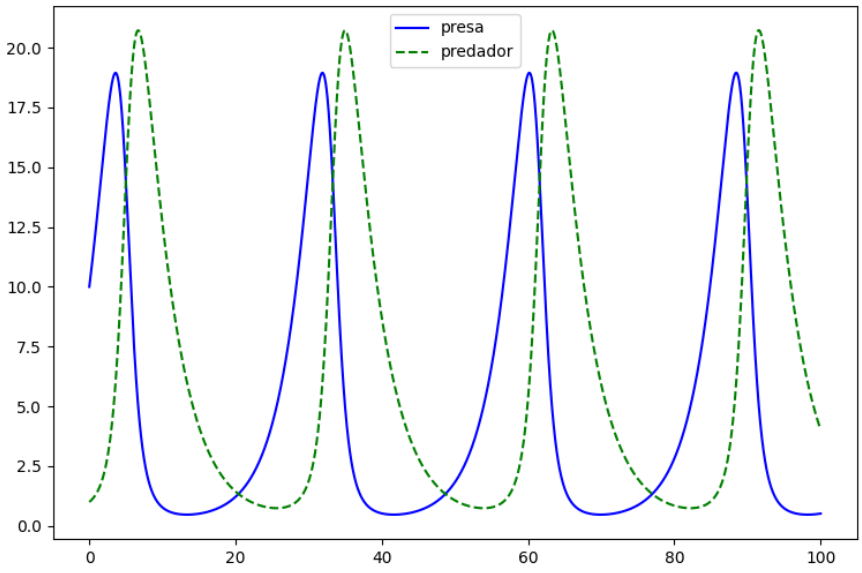

Figura 1 – Exemplo de resultado do modelo Predador-Presa clássico.

2.2 Programação Visual

Programação visual é uma maneira do usuário programar a máquina por meio de elementos gráficos que abstraem instruções do computador. Comumente, estes elementos representam múltiplas operações por vez, com o objetivo de facilitar a programação.

Um dos primeiros exemplos da aplicação prática da programação visual foi com a linguagem GRAIL (*GRAphical Input Language*) (ELLIS; HEAFNER; SIBLEY, 1969), desenvolvida em 1969 junto a um dispositivo semelhante a uma caneta e uma tela sensível ao seu toque. O conjunto do *hardware* e do *software* permitia que o usuário desenhasse letras e formas geométricas que se traduziriam em elementos de fluxograma. O objetivo era realizar um estudo sobre a comunicação humano-computador.

O conceito de utilizar elementos visuais para a assistência da programação não se restringiu somente a dispositivos especializados, contudo, e hoje se encontra em diversos *softwares* de finalidades distintas que podem ser instalados em qualquer computador. Como um exemplo de aplicação de propósito similar ao GRAIL, a linguagem Scratch (MALONEY et al., 2010) foi criada com o objetivo de facilitar o ensino de lógica de programação para crianças sem comprometer em funcionalidades. A linguagem apresenta comandos, funções e estruturas de controle como blocos que podem ser "montados" para compor um programa.

Outras aplicações se especializam em algum nicho da computação, como é o caso do simulador lógico Logisim (BURCH, 2002), que abstrai componentes lógicos e permite o usuário criar circuitos em uma espécie de *protoboard* digital.

2.2.1 Editores baseados em nós

Muitas vezes, *softwares* que incluem programação visual para a abstração de operações utilizam interfaces conhecidas como "editores baseados em nós", que são caracterizados pela presença de blocos flutuantes conhecidos como "nós".

Estes nós são responsáveis por criar ou manipular informações, que podem ser repassadas a outros nós, se assemelhando a uma esteira de produção. A passagem destas informações é feita por componentes contidos nos nós conhecidos como "pinos", que permitem a entrada ou saída dos dados. Comumente, o fluxo de passagem de dados ocorre da esquerda para a direita, sendo que os pinos à esquerda de um nó são os pinos de entrada e pinos à direita são os pinos de saída.

Alguns exemplos de *softwares* desse tipo são:

- O *software* de modelagem, animação e edição de vídeos, Blender (TAKALA; MäKäRäINEN; HAMALAINEN, 2013), utiliza um editor de nós para abstrair operações de manipulação de imagens e materiais;
- O motor de jogos proprietário Unreal Engine[1] utiliza editores de nós para múltiplas finalidades e, entre elas, está a escrita de *shaders* e a programação de lógica geral dos jogos.

[1] Documentação da Unreal Engine

Existem diversos motivos pelos os quais a programação visual se encontra em tão amplo uso. Principalmente, pode-se atribuir o sucesso de sua aplicação à facilidade de utilização. Em geral, não se faz necessário a leitura de manuais da mesma forma que se faz com linguagens de programação tradicionais; todas as operações possíveis no software estão modeladas como elementos visuais com parâmetros de entrada e saída bem definidos. Implementações exemplares também proibirão o usuário de realizar ligações inválidas diretamente na interface, realizando uma etapa mais comum em compiladores ou analizadores estáticos de código, que geralmente permitem que o usuário cometa um ou mais erros e só descubra-os posteriormente.

2.3 Geração de código

A fase de geração de código geralmente é a última fase da compilação. Na maioria dos compiladores, o gerador de código recebe como entrada uma ou mais representações intermediárias do código/texto de entrada e retorna como saída o código/texto na linguagem alvo (AHO et al., 2006).

Como este trabalho tem o objetivo de possibilitar a geração de código em múltiplas linguagens alvo diferentes, surgiu a necessidade de abstrair mais esta fase, de forma que a implementação de linguagens alvo adicionais seja mais simples considerando que ao menos uma já foi implementada.

Para tal abstração, seguiu-se um padrão utilizado por diversos compiladores diferentes, que consiste na separação do *back-end* (que trabalha com a linguagem alvo) do *front-end* (que trabalha com a linguagem fonte) através da utilização de uma ou mais representações intermediárias.

Nas próximas subseções, serão apresentados alguns conceitos importantes para o entendimento do processo de geração de código.

2.3.1 Representação intermediária

As representações intermediárias (RIs) são usadas em um compilador por várias razões (AHO et al., 2006), dentre as quais destaca-se:

- Separar o *front-end* do *back-end*. Neste caso, o *front-end* não precisa se preocupar com detalhes da linguagem alvo e o *back-end* não precisa conhecer detalhes da linguagem fonte;
- Permitir que sejam realizadas otimizações independente de máquina ou otimizações independente da linguagem alvo;
- Para facilitar a tradução e geração do código alvo.

Algumas RIs usadas em compiladores incluem:

- Árvore de Sintaxe Abstrata (ASA);
- Grafos Acíclicos Dirigidos (GAD);
- Código de Três Endereços.

As Árvores de Sintaxe Abstratas são muito usadas na representação do código fonte de entrada, representando comandos e expressões em cada escopo. As Árvores de Sintaxe Abstratas geralmente são utilizadas para realizar diversas verificações semânticas no código e elas podem ser a entrada para o algoritmo que gera uma outra RI mais próxima da linguagem alvo como, por exemplo, o Código de Três Endereços (AHO et al., 2006).

O Código de Três Endereços é uma representação que está próxima da linguagem Assembly e pode ser usada como base para várias otimizações independentes de máquina. Geralmente, o Código de Três Endereços otimizado é a entrada recebida pelo gerador de código Assembly. O Código de Três Endereços também pode ser utilizado pelo algoritmo de alocação de registradores como uma abstração inicial para a escolha e atribuição de registradores (AHO et al., 2006).

Os Grafos Acíclicos Dirigidos são muito usados para realizar algumas otimizações no código como, por exemplo, verificação de variável morta, propagação de constantes e cópias, entre outras otimizações (AHO et al., 2006).

Como uma alternativa mais abstrata ao Assembly (que é dependente da arquitetura alvo), existe a linguagem intermediária da LLVM (LLVM *Intermediate Representation*), que pode ser transpilada para o Assembly da máquina alvo. A LLVM apresenta uma infraestrutura para facilitar o desenvolvimento de compiladores abstraindo toda a parte do backend. O desenvolvedor não precisa se preocupar em conhecer e programar para as arquiteturas alvo com as quais quer trabalhar porque, considerando que a linguagem criada possa ser transpilada na representação intermediária da LLVM (LLVM IR), ela pode ser compilada em qualquer arquitetura suportada pela infraestrutura que incluem, mas não estão limitadas a, x86, AMD64, ARM, ARM64, WebAssembly[2] e RISC-V.

Para facilitar o processo de geração de código e realizar uma maior separação entre o *front-end* (interface gráfica) e o *back-end* (gerador de código) do software, foi criada e utilizada uma representação intermediária (RI). A RI é a entrada para o gerador de código e também é utilizada para salvar e carregar os modelos construídos no software. A RI desenvolvida será apresentada e explicada na Seção 4.1.2.

[2] WebAssembly Website

2.3.2 Sistemas baseados em *templates*

Em várias aplicações, a geração de código é feita com base em *templates*. Desde o início da *World Wide Web*, *templates* têm sido usados por *frameworks* para ajudar no processo de construção de páginas Web. Informalmente, podemos dizer que um *template* é um esqueleto (uma estrutura) que serve de referência para todo o processo de geração de código.

Em sua essência, *templates* são arquivos com marcadores especiais que podem ser substituídos por outros valores dinamicamente, de forma similar ao pré-processador de C. Adicionalmente, alguns motores de *templates* introduzem estruturas de controle de fluxo, como condicionais, laços de repetição e chamadas de funções para facilitar a injeção de código nos *templates*.

Para ilustrar a ideia, suponha o *template* abaixo criado com base em um código que implementa um sistema de Equações Diferenciais Ordinárias (EDOs):

```
1  from scipy.integrate import solve_ivp
2  import numpy as np
3  import pandas as pd
4
5  def odeSystem(t, P, {% for key, value in params %}{%if loop.is_last%} {{
       key}} {%else%} {{key}}, {%endif%}{%endfor%}):
6  ## for key, value in vars
7      {{key}} = P[{{loop.index}}]
8  ## endfor
9  ## for key, value in odes
10     d{{key}}_dt = {{value}}
11 ## endfor
12     return  [{% for key, value in odes %}{%if loop.is_last%} d{{key}}_dt
       {%else%} d{{key}}_dt, {%endif%}{%endfor%}]
```

O *template* acima utiliza os marcadores definidos pela biblioteca Inja[3], e demonstra algumas das estruturas de controle mencionadas, na forma das palavras-chave `for`, `if` e `else`.

2.4 Ensino-aprendizagem de Modelagem Computacional

Os modelos são abstrações do mundo real. Com a evolução dos computadores e o seu uso massivo, os modelos computacionais têm contribuído significativamente para o avanço do conhecimento em diversas áreas. Eles nos ajudam a compreender os fenômenos de interesse permitindo "rastrear" e entender as mudanças que estão ocorrendo em um sistema complexo e visualizar, por exemplo, o efeito de uma pequena mudança no restante do sistema (BARELLI, 2022).

[3] https://github.com/pantor/inja

Simulações computacionais são ferramentas fundamentais não só para a pesquisa científica, mas também para a educação. Elas são frequentemente usadas como laboratórios virtuais para promover a compreensão dos alunos sobre os conceitos teóricos que estão na base dos sistemas simulados (BARELLI, 2022).

Após a Segunda Guerra Mundial, o leque de campos disciplinares em que as simulações começaram a ser utilizadas tem vindo a expandir-se até os dias de hoje, de modo que é quase impossível nomear qualquer disciplina que não tenha utilizado ou desenvolvido ferramentas computacionais e simulações para avançar a fronteira do conhecimento (BORRELLI; WELLMANN, 2019).

O uso de modelos computacionais abre novas maneiras para os alunos aprenderem sobre diferentes disciplinas. Os alunos podem aprender através de um processo iterativo de construção do modelo, simulação, modificação do modelo para simulação novamente, explorando todas as características do modelo em si (BARELLI, 2022). Plataformas como, por exemplo, a Netlogo (WILENSKY, 2015) permitem este tipo de aprendizagem iterativa e interativa.

Um dos objetivos do software que foi desenvolvido neste trabalho é auxiliar o aprendizado de conceitos importantes nas áreas de modelagem matemática e computacional e ajudar o aluno a desenvolver a habilidade de modelar problemas através da prática de construção e simulação de modelos. Acredita-se que o uso combinado do software com alguma metodologia ativa de ensino permitirá aos alunos aprender na prática como construir modelos para diferentes fenômenos de interesse e entender melhor os fenômenos estudados através das simulações computacionais realizadas de forma interativa através da interface gráfica do software.

Considerando-se o uso do software na prática e uma visão geral do chamado ciclo da modelagem apresentado na Figura 2, destaca-se que o software auxilia o passo 2 e automatiza os passos 3 e 4 facilitando o processo de modelagem por estudantes e pesquisadores.

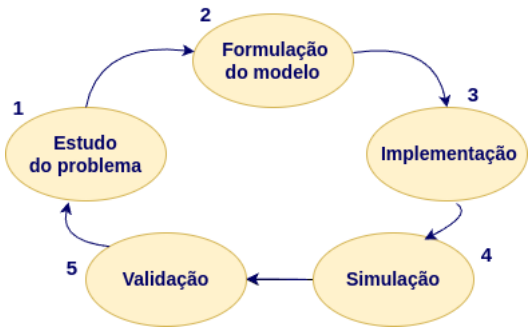

Figura 2 – Visão geral do ciclo da modelagem.

O aprendizado, em sala de aula, pode ser conduzido, por exemplo, através de um processo incremental que começa investigando e trabalhando com modelos mais simples

até se chegar em modelos mais complexos. Algumas vantagens dessa abordagem incluem: 1) a apresentação de determinados conceitos e técnicas é facilitada em modelos mais simples; 2) em muitos casos reais, modelos complexos utilizam certas expressões, equações e "estratégias" na modelagem que estão presentes em modelos mais simples. Então, muitas vezes, o aluno conseguirá entender melhor um modelo mais complexo entendendo que determinadas partes do mesmo já apareceram em modelos mais simples previamente estudados.

Uma possível abordagem de ensino seria o professor trabalhar com algum problema escolhido por ele para apresentar e discutir conceitos e/ou técnicas de modelagem de forma semelhante à metodologia *Problem Based Learning* (PBL) (AKCAY, 2009). O seguinte conjunto de passos é sugerido para o processo de estudo, modelagem e simulação computacional:

1. Descrição do contexto e do problema a ser modelado;
2. Formulação das hipóteses pelos alunos e discussão entre eles;
3. Formulação do modelo considerando o conhecimento de modelos existentes e que foram estudados previamente;
4. Construção do modelo no software;
5. Simulação do modelo no software com a criação de diferentes cenários para teste e análise dos comportamentos obtidos;
6. O professor pode fazer perguntas sobre o modelo e pedir para os alunos tentarem responder através das simulações;
7. Avaliação dos resultados do modelo com a ajuda do professor: caso o comportamento desejado ou esperado não seja obtido, voltar para um dos passos anteriores como, por exemplo, o passo 2 ou 3 e repetir o processo.

A ideia dessa abordagem é colocar o aluno como protagonista no processo de construção do próprio conhecimento porque o aluno estará ativamente realizando todas as etapas de modelagem do problema (AKCAY, 2009; BARELLI, 2022).

Com o uso do software, acredita-se que os alunos poderão investir mais tempo formulando as hipóteses e equações do modelo, planejando os cenários a serem simulados e avaliando os resultados obtidos. Isso porque o software removerá a necessidade de aprender linguagens de programação e bibliotecas de simulação, que mesmo depois de aprendidas podem gerar complicações, como erros de compilação ou o levantamento de erros em tempo de execução. Ao invés, os alunos poderão aprender a utilizar o software, que apresenta

um número reduzido de conceitos e opções de interação, impossibilitando a construção de modelos inválidos.

3 Trabalhos relacionados

Com o avanço de tecnologias para o desenvolvimento de aplicações gráficas e também da internet, ocorreu um aumento expressivo na quantidade de *softwares* desenvolvidos para modelagem e simulação computacional.

Na Tabela 1 é apresentado um quadro comparativo entre o software deste trabalho e softwares similares. Para realizar a comparação, foram selecionadas algumas características consideradas relevantes no contexto de softwares para modelagem e simulação computacional.

		Acesso		Funcionalidades		
Software	Modelos	Distribuição	Open Source?	Representação Gráfica?	Exporta Código?	Simulação Interativa?
ODE Designer	EDOs	Desktop com dependências inclusas	Sim	Sim	Sim	Sim
Snoopy	Redes de Petri	Desktop com dependências inclusas	Não	Sim	Não	Sim
VCell	EDOs, EDPs e modelos Estocásticos	Desktop com dependências inclusas	Sim	Sim**	Não	Sim
InsightMaker	EDOs, Dinâmicas de Sistemas, Modelos baseados em Agentes	Aplicação Web	Não*	Sim**	Não	Sim
Cytoscape	Redes Complexas	Desktop	Sim	Sim	Não	Sim
EpiFire	Redes de Contatos (*Contact Networks*)	Desktop com dependências inclusas	Sim	Não	Não	Sim

Tabela 1 – Tabela comparativa entre os principais softwares de modelagem computacional e o software desenvolvido neste trabalho.
*Apenas a biblioteca de simulação é *open source*, não incluindo a interface gráfica.
**Os modelos são escritos em formas de equações e não necessariamente serão representados fielmente pela representação gráfica destes softwares.

O software desenvolvido neste trabalho possui alguns diferenciais em relação a outros desenvolvidos com propósitos similares:

- Em sua interface gráfica, é apresentado um editor baseado em nós que fornece ao usuário uma forma de programação visual que facilita a construção e simulação de modelos, limitando apenas operações consideradas incorretas;
- Em seu formato de representação intermediária para salvamento e carregamento dos modelos, o software utiliza uma estrutura extensível para outros tipos de modelos como, por exemplo, equações diferenciais parciais. Esta estrutura também foi projetada para ser humanamente legível, a ponto de ser possível que uma pessoa decodifique o modelo representado apenas lendo o arquivo. Assim, caso o software se prove insuficiente ou indisponível, ainda é possível reverter o arquivo nas equações que o compõe.
- O software permite ao usuário acessar o código-fonte usado para simular o modelo, possibilitando que o usuário conheça o código usado na simulação e também possa utilizar e alterar o código como desejar.

Poucos *softwares* encontrados na literatura permitem que o usuário visualize e interaja com o código gerado para a simulação de seu modelo. Mais comumente, a única maneira com a qual o usuário pode interagir com o modelo é através da abstração provida pela interface, o que causa uma dependência direta do *software*.

A seguir, são descritos em mais detalhes os trabalhos relacionados.

3.1 Snoopy

Um *software* que serviu como inspiração para este trabalho foi o software para construção, animação e simulação de redes de Petri chamado Snoopy (HEINER; RICHTER; SCHWARICK, 2008; HEINER et al., 2012; LIU, 2012). Além da rede de Petri clássica, o software permite modelar e simular vários tipos de redes de Petri como, por exemplo, redes de Petri estocásticas e coloridas que são extensões da rede clássica. O modelo construído é representado como um grafo que tem dois tipos de nós chamados *places* (locais) e *transitions* (transições) e quatro tipos de arestas (estocástica, imediata, determinística e planejada) com suas semânticas associadas. Os *places* representam populações de um certo tipo e transições representam eventos que podem aumentar ou diminuir as quantidades das populações que estão armazenadas nos locais.

O modelo gráfico utilizado pelo software Snoopy serviu de inspiração para a criação da representação visual utilizada neste trabalho. A partir do software Snoopy, foi possível observar que modelos gráficos, além do atrativo visual, facilitam a construção do modelo,

e um melhor entendimento do que está sendo feito e o que está sendo modelado. A possibilidade de simular interativamente as Redes de Petri estocásticas foi um fato que serviu de inspiração para a ideia da simulação interativa no software desenvolvido neste trabalho.

3.2 Insight maker

O InsightMaker (FORTMANN-ROE, 2014) é um *software* de simulação baseado em nuvem. Todo o modelo é feito através de um navegador de internet e, quando executado, é simulado no próprio servidor, sem a necessidade de que o usuário possua uma máquina potente. O InsightMaker (FORTMANN-ROE, 2014) permite a construção de modelos utilizando diagramas de Dinâmica de Sistemas e modelos baseados em agentes. O site possui diversos recursos para a construção do modelo em um Canvas. A partir do modelo de Dinâmica de Sistemas construído no Canvas, o InsightMaker (FORTMANN-ROE, 2014) gera internamente um modelo de EDOs e executa este modelo utilizando o método de Euler ou o método de Runge-Kutta de quarta ordem conforme escolha do usuário. Através da interface web, também é possível definir os valores de todas as variáveis e parâmetros, e é possível realizar as simulações computacionais. Os resultados são exibidos de forma interativa onde é possível controlar a velocidade com que os valores são exibidos nos gráficos.

3.3 VCell

O software VCell (Virtual Cell) (MORARU et al., 2008; SCHAFF et al., 1997) é uma plataforma para modelar sistemas biológicos celulares que é construída em torno de um banco de dados central e disseminada como um aplicativo da web. Os modelos são construídos com base em regras (*Rule-based modelling*). VCell tem os seguintes tipos de simulações disponíveis: determinística (EDO compartimental ou EDP de reação-difusão-advecção com suporte para cinemática 2D), reações estocásticas (solvers de SSA), estocástica espacial (reação-difusão com Smoldyn), híbrida determinística/estocástica e simulações baseadas em agentes. O software permite simular vários processos metabólicos que ocorrem dentro das células e através das membranas celulares. Com relação às membranas, há suporte para os seguintes processos: Eletrofisiologia, suporte para o fluxo da membrana e difusão lateral da membrana.

Durante alguns testes com o VCell (MORARU et al., 2008; SCHAFF et al., 1997), sentiu-se a necessidade da leitura de um manual dado o fato de que o software apresenta muitas opções nas diferentes telas que são abertas para a construção de um modelo de EDOs. Durante os testes, não foi possível entender como o modelo de EDOs é construído e qual é o sistema de equações final que é usado na simulação. Acredita-se que isso tenha

ocorrido devido ao software apresentar muitas telas com muitas opções diferentes o que torna difícil entender e navegar pelas diferentes opções.

3.4 EpiFire

O EpiFire (HLADISH et al., 2012) não é um *software* completo, mas uma biblioteca em C++ para simulação de modelos de redes complexas com aplicação na área de Epidemiologia. Ele é uma interface de programação de aplicativos (API) implementado em C++, projetado para gerar de forma eficiente redes com uma distribuição de grau especificada, medir características fundamentais da rede e realizar simulações computacionais eficientes das redes geradas com base nos métodos *percolation* e *chain-binomial*.

3.5 Cytoscape

O software Cytoscape (SHANNON et al., 2003) é uma plataforma de software de código aberto para visualizar redes de interação molecular e vias biológicas e integrar essas redes com anotações, perfis de expressão gênica e outros dados. O software permite carregar conjuntos de dados de interação molecular e genética em vários formatos padrão. O sistema de *plugins* permite a extensão do software através de diversas aplicações desenvolvidas seguindo uma API baseada no modelo OSGi. O Cytoscape também é utilizado como uma plataforma geral para análise e visualização de redes complexas.

4 Software para modelagem e simulação computacional

O *software* desenvolvido neste trabalho consiste em uma interface gráfica de usuário (mais comumente referida pelo acrônimo em inglês, GUI, que significa *Graphical User Interface*) que apresenta um editor baseado em nós. Os nós representam abstrações de componentes comumente usados na construção de EDOs, como constantes e variáveis. A Figura 3 apresenta o editor de nós do software contendo o modelo Predador-Presa. Na Seção 4.1, serão discutidos os tipos de nós presentes na interface.

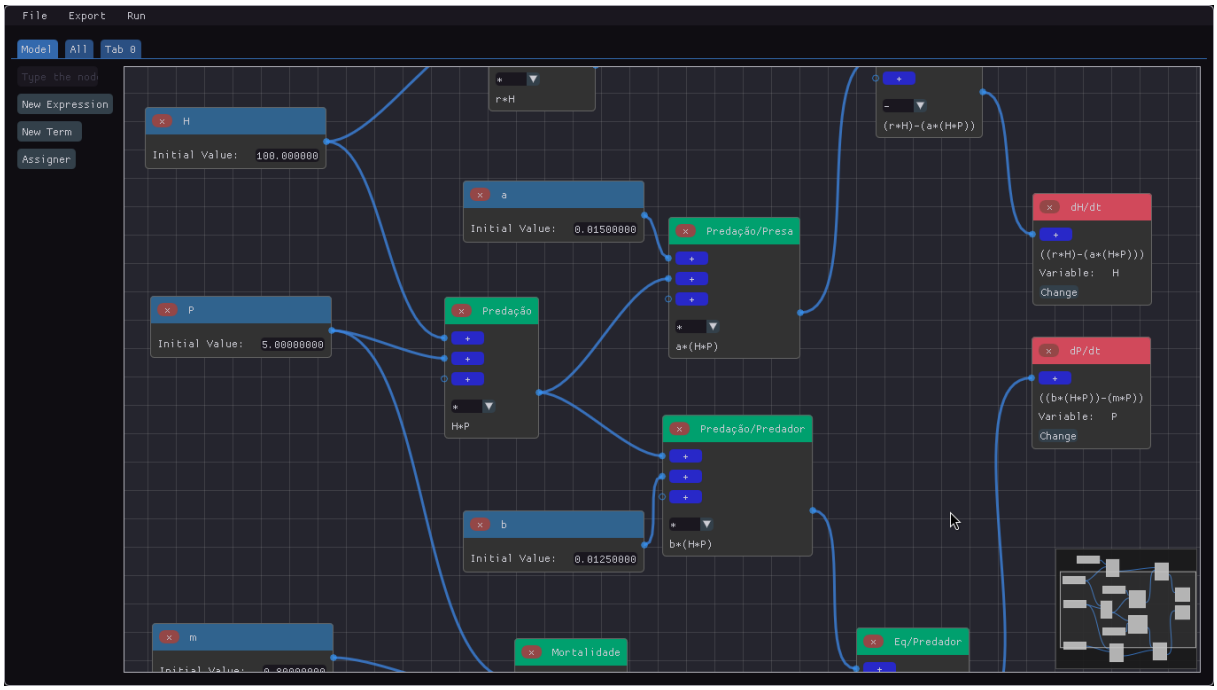

Figura 3 – Exemplo do modelo Predador-Presa (Equação 2.1) construído no *software*.

O modelo é construído a partir da criação de nós e suas ligações. Uma Equação pode ser expressada por um ou mais termos e combinações, e uma variável alvo para a atribuição do valor.

Após construído o modelo, o usuário poderá simulá-lo diretamente pela GUI (demonstrado na Figura 4), exportar um PDF dos resultados, ou exportar um código de Python equivalente para o modelo desenhado (demonstrado pelo anexo .1, que contém o código do modelo Predador-Presa). A experiência de usuário pode ser resumida ao fluxograma contido na Figura 5.

Dessa forma, o usuário pode criar e simular modelos sem a necessidade de iniciar a implementação computacional do zero e também sem a necessidade de ter conhecimentos

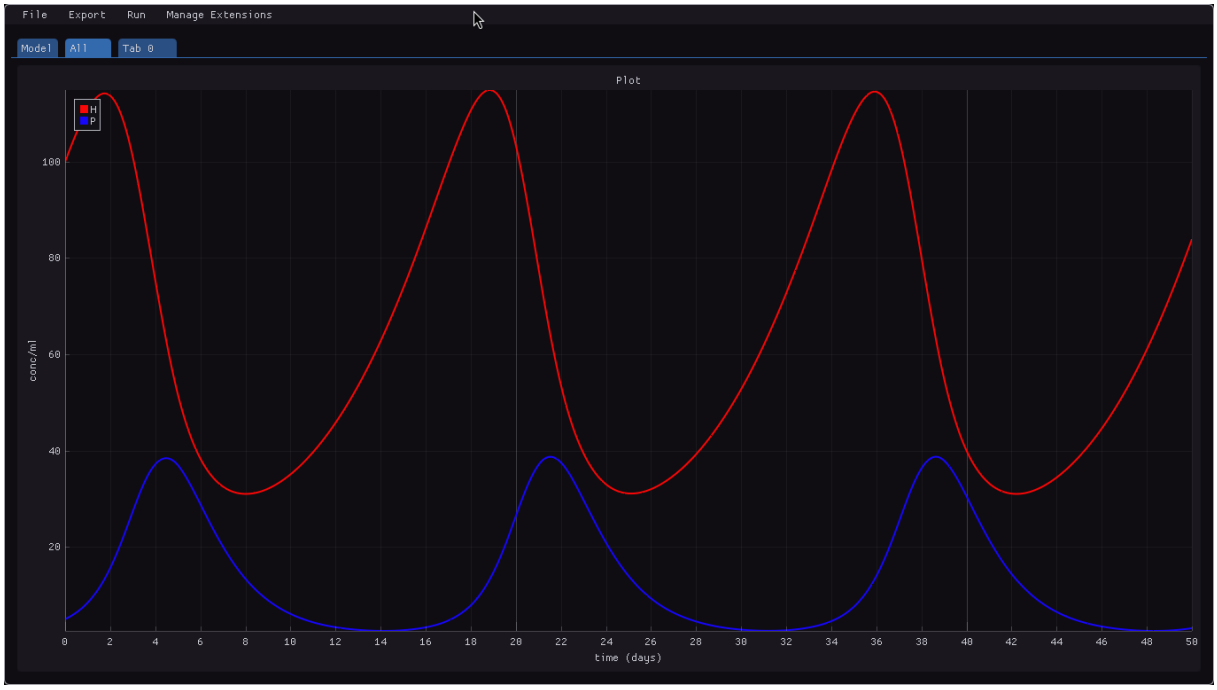

Figura 4 – Plotagem dos resultados da simulação do modelo Predador-Presa.

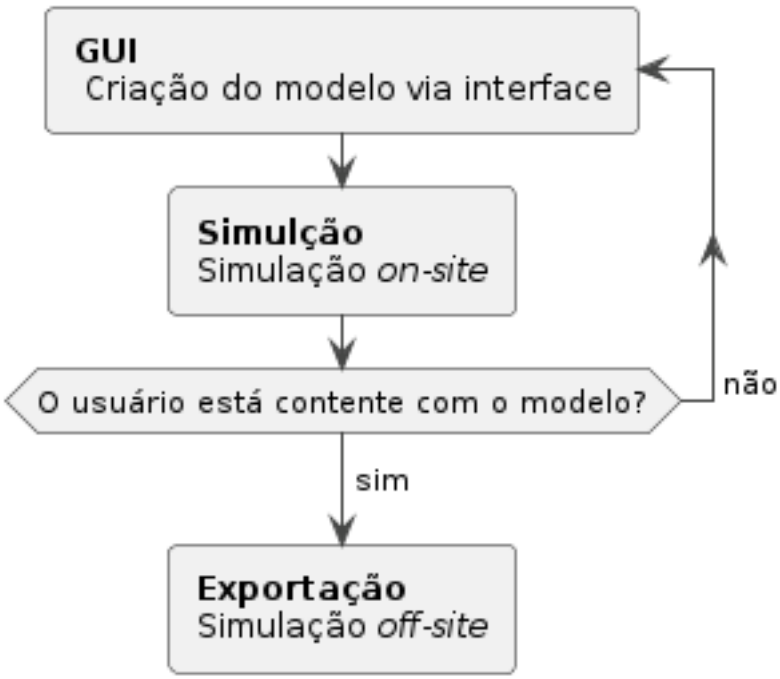

Figura 5 – Fluxograma da experiência de usuário esperada para o software.

dos métodos que são utilizados na implementação. Pelo contrário, o usuário terá acesso aos códigos gerados e poderá aprender com estes e utilizá-los, por exemplo, como base para desenvolver o seu próprio código.

A simulação das EDOs, tanto pela GUI quanto pelo código de Python exportado, é feita utilizando métodos especializados da biblioteca científica `scipy` (VIRTANEN et al., 2020).

Tipo de nó	Usado para representar	Exemplo
Termo	Variáveis, parâmetros e constantes.	Em uma expressão $k.x.y$, os termos são o parâmetro k e as variáveis x e y.
Expressão	Qualquer expressão matemática para a construção das equações do modelo.	Na Equação da presa (contida na Equação 2.1), as expressões são: 1) $r.H$; 2) $a.H.P$; e 3) $r.H - a.H.P$ (combinação de 1 e 2).
Equação	É um nó especial usado para atribuir uma expressão, que representa todo o lado direito de uma EDO, a uma variável.	A expressão $r.H - a.H.P$ pode ser usada como entrada de um nó de Equação associado à variável H.

Tabela 2 – Tipos de nós presentes no software.

4.1 Uma representação visual para EDOs

Uma representação visual foi desenvolvida para minimizar a necessidade de um conhecimento técnico prévio pelo usuário, assim como a dependência de documentos e manuais. Para atingir este objetivo, foi necessário modelar esta representação utilizando conceitos que fossem naturais para o público-alvo e a forma com a qual este desenvolve EDOs.

A construção de EDOs se baseia em alguns princípios gerais, como a Lei de Ação de Massas. Por exemplo, dadas duas populações A e B, caso seja desejado simular uma condição que seja mais provável de acontecer dado altas concentrações de ambas as populações, é possível representá-la simplesmente multiplicando ambas as populações. Opcionalmente, adiciona-se uma constante para controlar a taxa de ocorrência.

No sistema de EDOs na Equação 2.1, é possível visualizar este fenômeno: na Equação da presa, a expressão $-a.H.P$ é utilizada para controlar a taxa de predação, isto é, o quanto a concentração desta população é afetada negativamente pela atividade de predação realizada pelos predadores. Naturalmente, a predação ocorrerá mais frequentemente dado altas concentrações de ambos presa e predador, assim como descrito acima.

A partir desta premissa, pode-se resumir o desenvolvimento de EDOs à combinação das expressões que representam as interações das populações entre si e o meio. Assim, a representação visual foi modelada em um editor de nós, de forma que todas as interações visassem a construção destas expressões. A Tabela 2 descreve os tipos de nós presentes no software e sua relação com o objetivo de construir as expressões. A Figura 6 mostra os respectivos nós da Tabela 2 construídos na GUI do software.

A disposição dos pinos nos nós presentes foi planejada para minimizar a necessidade de manuais e permitir que o usuário aprenda a usar o software por conta própria. Isto porque:

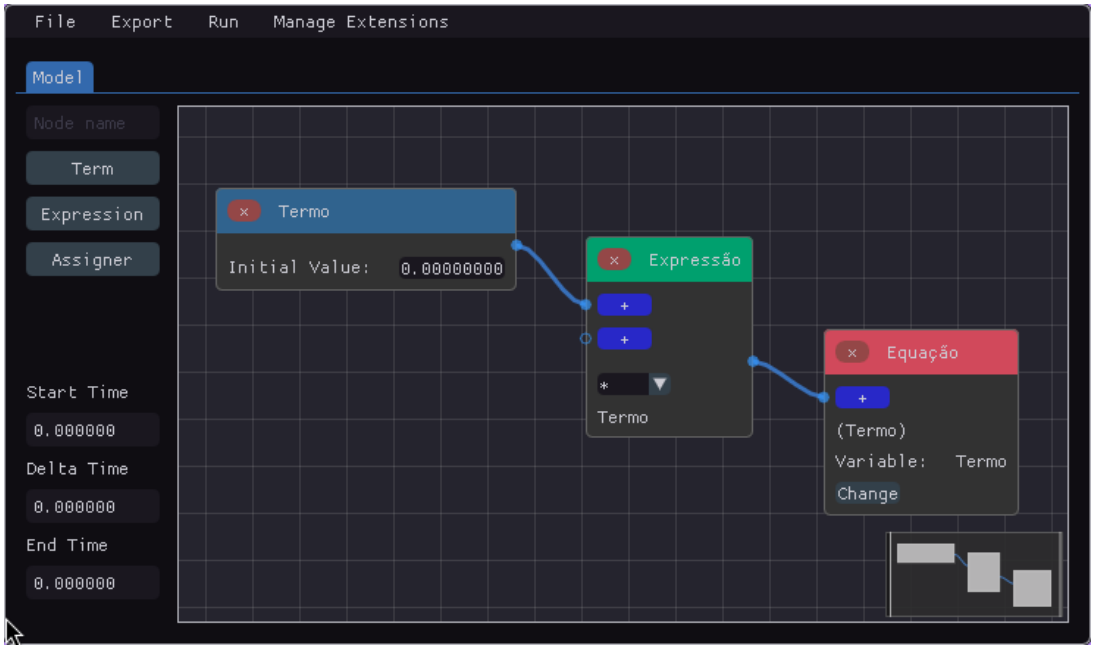

Figura 6 – Visualização dos nós Termo, Expressão e *Assigner* (ou Equação) no editor baseado em nós do software. Um exemplo de uso desses nós foi apresentado na Tabela 2.

- O nó 'Termo' possui um pino de saída e nenhum pino de entrada;
- O nó 'Expressão' possui N pinos de entrada e 1 pino de saída;
- O nó 'Equação' possui 1 pino de entrada e nenhum pino de saída.

Desta forma, é possível concluir que:

- Termos podem ser ligados a expressões e equações. Como não recebem nenhuma ligação, são os nós iniciais;
- Expressões podem ser ligadas a outras expressões e equações;
- Equações são os nós terminais.

4.1.1 Representação de expressões na interface

Na GUI, os nós de expressão apresentam a expressão que estão construindo, para que o usuário não tenha que inspecionar cada nó individualmente para determinar como uma expressão está sendo formada. Desta forma, o usuário pode ver a construção de todas as expressões que formam as EDOs e não apenas a equação final. Isto é feito por meio de árvores de expressões que cada um desses nós carrega consigo.

Na Figura 7, são apresentadas as árvores de expressões de todos os nós representados na interface pela Figura 3. Note que os nós 'Predação/Presa' e 'Predação/Predador'

referenciam o mesmo nó base 'Predação'. Isto demonstra que expressões podem ser usadas como entradas para outras expressões na forma de subexpressões. As subexpressões, além de possibilitar o usuário de evitar repetições na construção de modelos, permitem que os nós façam alterações relevantes para o contexto, como a adição de parênteses nas operações de multiplicação e divisão.

Esta representação foi desenvolvida de maneira agnóstica ao modelo, com o objetivo de permitir que o software seja extensível e, no futuro, oferte outros tipos de modelos. Um exemplo deste esforço se dá pela árvore de expressões não supôr que toda operação é infixada, também permitindo operações pós-fixadas. Exemplos da extensibilidade serão apresentados na Seção 4.3, enquanto os trabalhos futuros serão discutidos no Capítulo 6.

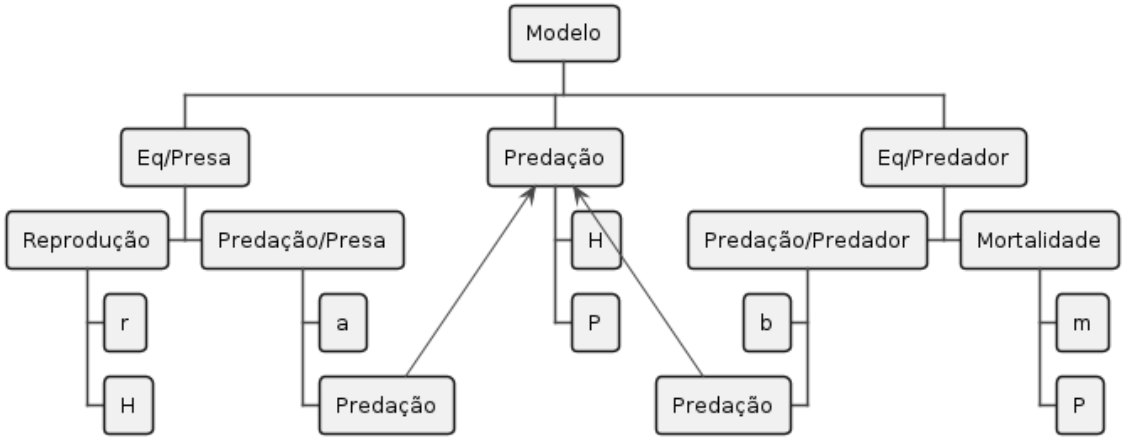

Figura 7 – Árvores de expressões para cada nó de expressão no modelo predador-presa, apresentado na Figura 3.

4.1.2 Uma representação intermediária humanamente legível

Com a finalidade de possibilitar a escrita e leitura em disco dos modelos desenhados na GUI, uma RI foi definida. O formato JSON foi escolhido por sua popularização com a *web* e com sua clara representação de estruturas de dados como listas e tabelas *hash* que facilitam o desenvolvimento do software. A escolha de um formato pré-estabelecido também torna a RI mais resiliente a futuras mudanças, uma vez que não será necessário modificar a sintaxe do formato e nem as bibliotecas que o interpretam.

Apesar de ser usada como forma de armazenar informações do modelo, a RI possui um formato projetado para ser humanamente legível. Esta decisão foi tomada para possibilitar a recuperação do modelo construído sem a necessidade de ter o software instalado. Assim, caso o software caia em desuso, se torne impossível de instalar por algum motivo (um sistema operacional diferente, por exemplo) ou simplesmente tenha mudanças que quebrem compatibilidade implementadas, não significa que o usuário tenha perdido seu modelo.

Na Listagem 4.1, pode-se observar o trecho da RI do modelo Predador-Presa (RI completa no apêndice .2) que implementa a Equação da presa. Note que é possível reconstruir a Equação manualmente: cada entrada em `"arguments"` representa um termo (representado pelos campos nome e valor) ou uma combinação (representada pelos campos nome, operação e composição). Com este último, uma expressão pode ser construída a partir da repetida concatenação de todos os elementos de sua composição, inserindo a operação entre eles. Seguindo esta regra, tem-se as seguintes expressões:

- `"Predacao/Presa"`: $(+a) * (+H) * (+P)$, que simplificando se resume a $a * H * P$;
- `"Reprodução"`: $(+r) * (+H)$, que simplificando se resume a $r * H$;
- `"Eq/Presa"`: A subtração de `"Predacao/Presa"` e `"Reprodução"`, que resulta em $r * H - a * H * P$;

```
"arguments": [
    { "name": "Eq/Presa", " operation ": "-",
        " composition ": [
        { "name": "Reproducao", " contribution ": "+" },
        { "name": "Predacao/Presa", " contribution ": "+" }
    ]},
    { "name": "Reproducao", " operation ": "*",
        " composition ": [
        { "name": "r", " contribution ": "+" },
        { "name": "H", " contribution ": "+" }
    ]},
    { "name": "Predacao/Presa", " operation ": "*",
        " composition ": [
        { "name": "a", " contribution ": "+" },
        { "name": "H", " contribution ": "+" },
        { "name": "P", " contribution ": "+" }
    ]},
    { "name": "H", " value ": 100.0 },
    { "name": "P", " value ": 5.0 },
    { "name": "r", " value ": 0.2 },
    { "name": "a", " value ": 0.015 }
]
```

Listing 4.1 – Trecho da RI que define termos e combinações usadas para definir a Equação da presa.

4.2 Geração de Código e Simulação interativa

Além de apresentar uma interface capaz de criar, salvar e carregar sistemas de EDOs, o software também possibilita ao usuário simular o modelo (plotando os resultados na interface), gerar o código equivalente em Python ou exportar um PDF com os resultados.

A geração de código é feita por meio da injeção de informações do modelo em marcações presentes no *template* e se baseia na substituição de *strings*. Os *templates* incluem o código base necessário para a resolução numérica, simulação e plotagem dos resultados com a inclusão de todas as bibliotecas necessárias. Os templates são escritos de maneira sucinta e legível com o objetivo de servir como um material de aprendizagem estensível.

Na listagem de código 4.2, observa-se parte do template que extrai o valor de variáveis e constantes para serem usados na simulação do próximo passo de tempo. Neste trecho, nota-se o uso de estruturas de controle como condicionais (`if`) e laços de repetição (`for`). Baseando a geração de código no sistema de *templates*, é possível adicionar novas formas de simulações sem alterar o código do programa. Por exemplo, é possível adicionar um *template* para gerar o código que simulará o modelo de forma estocástica usando o algoritmo de Gillespie.

```
23 def system(t: np.float64, y: np.ndarray, *constants) -> np.ndarray:
24     {% for arg in populations -%}
25     {{- arg.name }},
26     {%- endfor %} = y
27 
28     {%- if constants %}
29     {% for arg in constants -%}
30     {{- arg.name }},
31     {%- endfor %} = constants
32     {% endif -%}
33     # ...
```

Listing 4.2 – Trecho do *template* responsável por extrair o valor de variáveis e constantes

O código gerado, além de ser apresentado como opção de exportação do modelo para o usuário, também é utilizado internamente pelo software para produzir os resultados que são plotados na interface. Como benefício desta abordagem, os gráficos plotados tanto pelo software quanto pelo código exportado são idênticos. Por outro lado, a dependência de Python para a simulação traz complicações quanto à distribuição do software, que será discutida em mais detalhes na Seção 4.5, assim como será discutida a solução implementada.

```python
import math

@node
def sin(x):
    return math.sin(x)

@node(format='$1 ^ $2')
def pow(x, y):
    return x ** y
```

Listing 4.3 – Exemplo de definição dos nós de extensão seno e potência com código Python válido.

4.3 Extensibilidade

É improvável que seja possível construir um software que atenda todas as necessidades de um público-alvo, e o software desenvolvido neste trabalho não é exceção; seria difícil determinar quais são as funcionalidades mínimas necessárias para construir qualquer sistema de EDOs. Visando, então, maximizar os casos de uso aos quais este software atende, foi desenvolvido um sistema de extensões de usuário.

Além de prover alguns nós padrões para a construção das EDOs, o software permite que o usuário desenvolva e utilize extensões para realizar operações que não estão prontamente disponíveis pela interface. Estas extensões são desenvolvidas em Python e incluem funções que serão interpretadas como nós na interface.

As funções decoradas por `@node` num arquivo de extensões serão interpretadas como nós. Seus nomes são usados em sua identificação na interface, e a quantidade de parâmetros que recebem determinam a quantidade de pinos de entradas. Por padrão, estes nós 'customizados' construirão expressões mais próximas da representação destas funções em linguagens de programação (como `funcao(param1, param2, ...)`), porém este comportamento pode ser sobrescrito com o parâmetro `format` no decorador.

Na listagem 4.3, é apresentado um exemplo de arquivo de extensões contendo um nó para calcular o seno de um parâmetro e um nó que eleva um parâmetro ao valor de outro. Este arquivo pode ser importado na interface, que possibilitará o uso dos nós como na Figura 8.

Como estes nós são definidos utilizando Python, não há impacto na capacidade do software de simular o modelo e plotar os gráficos; as funções definidas nas extensões são diretamente incluídas no código final que será utilizado pela simulação. Ademais, os modelos salvos pela interface mantém uma referência aos arquivos de extensões utilizados na construção do modelo, tornando possível que o software auto-importe as extensões necessárias quando carregar um modelo do disco. Isto permite que usuários não-técnicos utilizem extensões desenvolvidas por outras pessoas.

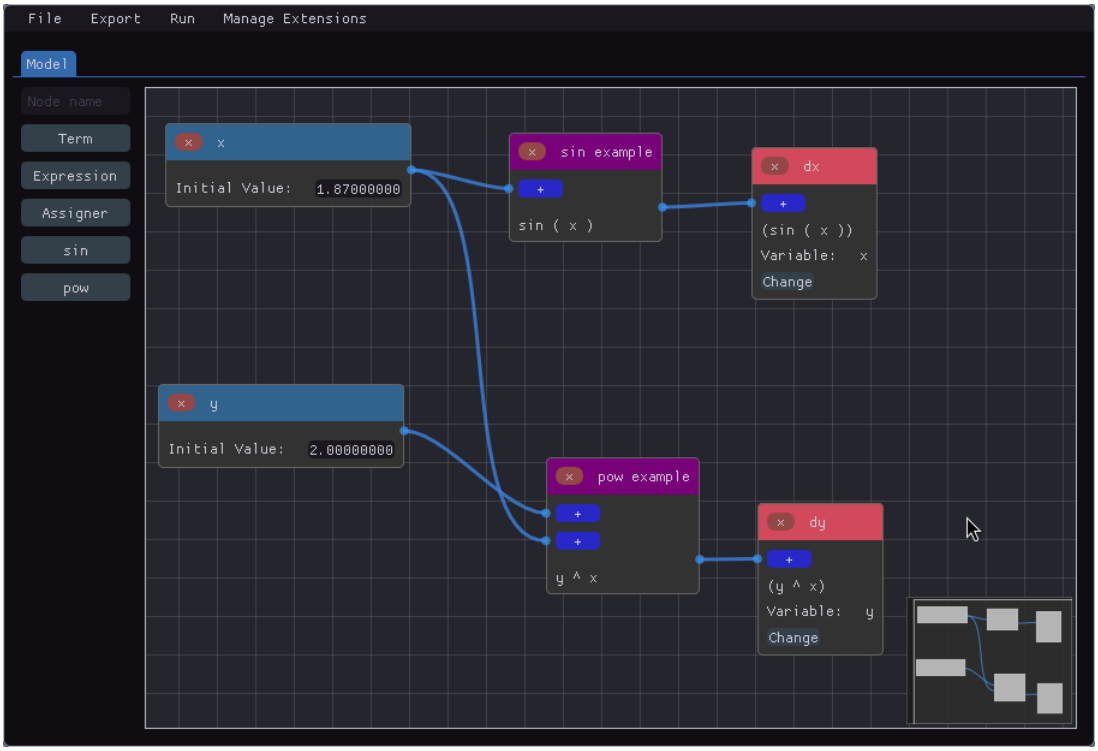

Figura 8 – Os nós definidos na listagem 4.3 sendo utilizados na interface.

4.3.1 Carregamento de extensões

Como mencionado anteriormente, os nós de extensões são definidos por funções que utilizam um decorador específico. Durante o carregamento de um arquivo de extensões, o software injeta um trecho especial de código que define o decorador `@node` para inspecionar a função que está sendo decorada e emitir um JSON com as informações relevantes sobre a função como nome, quantidade e nome dos parâmetros e o formato da representação na interface. Essas informações servem como especificações para novos construtores de nós.

4.4 Implementação e Tecnologias utilizadas

O software desenvolvido foi dividido em dois módulos: (1) o responsável pela interface gráfica e (2) o responsável pela representação intermediária. O primeiro lida com a apresentação de gráficos, interações com o usuário e a plotagem dos resultados, enquanto o segundo define a representação em JSON dos modelos e realiza a transformação em código Python.

4.4.1 O módulo da GUI

O módulo da GUI, desenvolvido em Rust, é a aplicação principal, e é responsável por toda a jornada do usuário. Este módulo inclui o editor de nós, o plotador de gráficos e as operações que envolvem a invocação do módulo da RI e a apresentação dos resultados

destas chamadas.

Entre seus submódulos principais, estão:

- O submódulo da aplicação, que renderiza o editor de nós e gerencia o estado global. A aplicação também controla as janelas auxiliares que podem ser abertas, como as abas de plotagem de resultados, o gerenciador de extensões e o atribuidor de variáveis. A Figura 9 apresenta o diagrama de classes desse módulo;

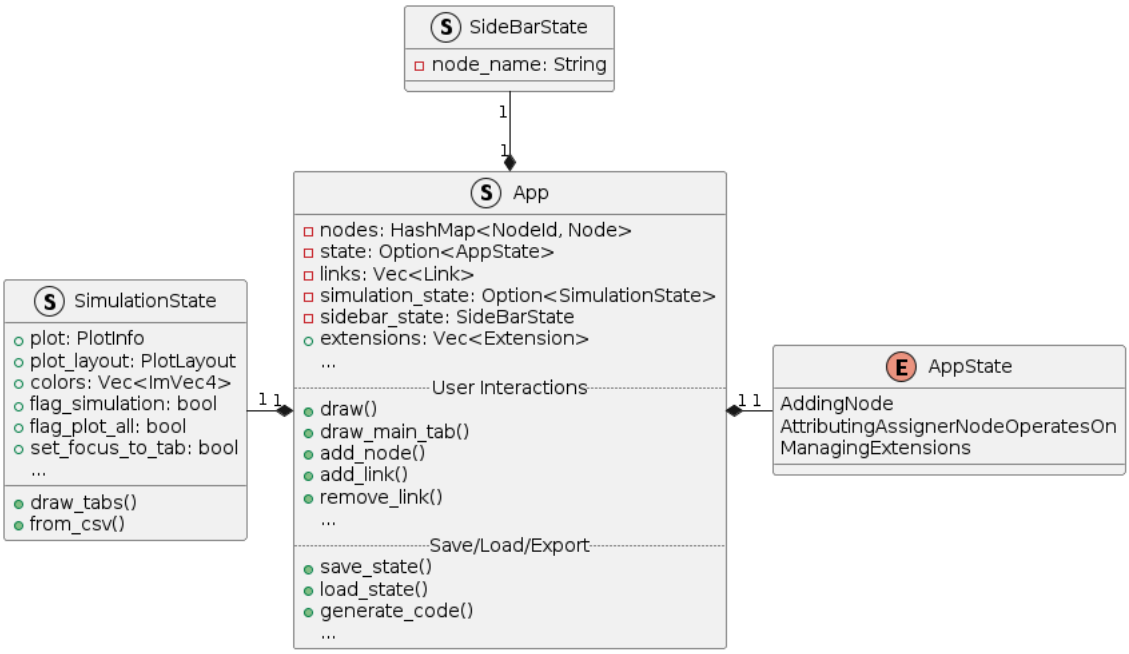

Figura 9 – Diagrama de classes do módulo da aplicação.

- O submódulo de nós, que define os possíveis tipos de nós. Estes possuem métodos para receber e enviar mensagens (notificando mudanças de estados, como alteração na árvore de expressão) e definir suas cores na interface. A Figura 10 apresenta o diagrama de classes desse módulo;

- O submódulo de extensão, que define as estruturas necessárias para analisar e carregar a definição de nós customizados em forma de funções de Python. A Figura 11 apresenta o diagrama de classes desse módulo;

- O submódulo de expressão, que define a árvore de expressões que representa o modelo e as estruturas que podem ser usadas durante a passagem de mensagens para comunicar mudanças parciais. A Figura 12 apresenta o diagrama de classes desse módulo;

- O submódulo de mensagens, que implementa a estrutura utilizada pelos outros módulos durante a comunicação intra-aplicação. A Figura 13 apresenta o diagrama de classes desse módulo.

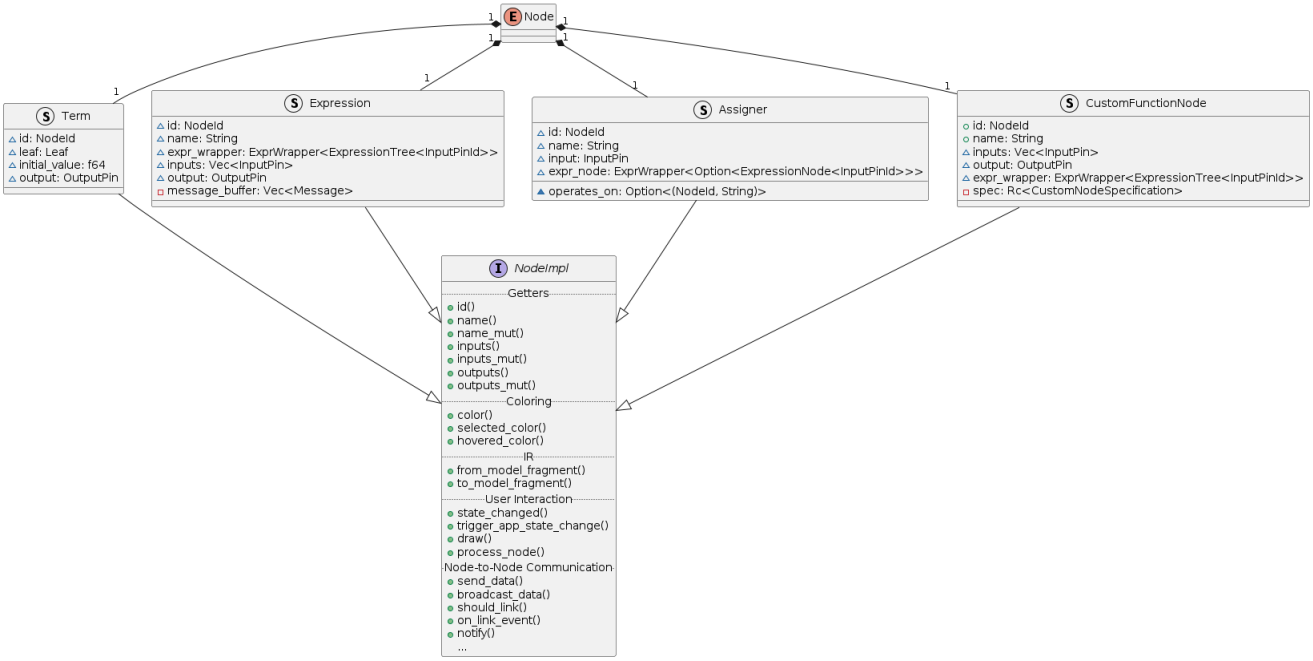

Figura 10 – Diagrama de classes do módulo de nós.

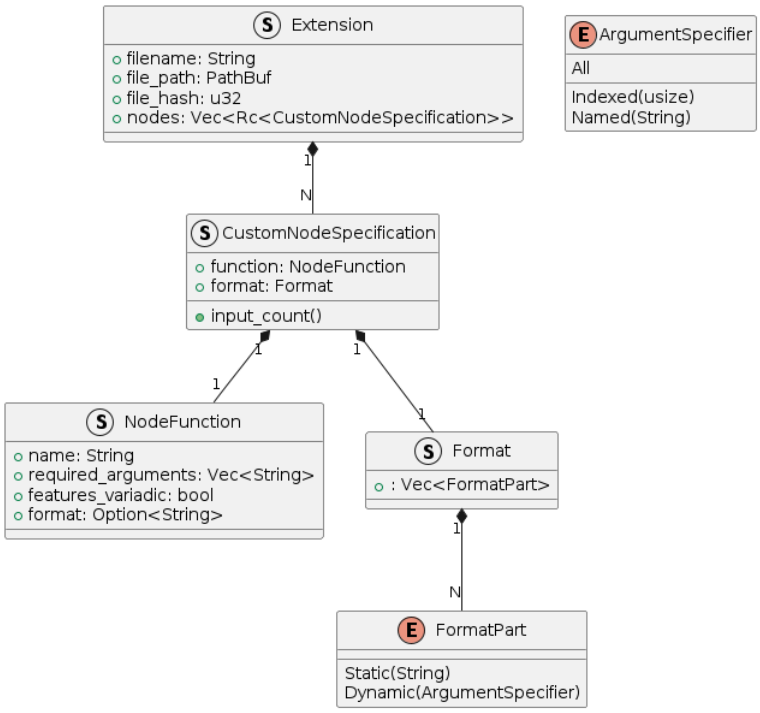

Figura 11 – Diagrama de classes do módulo de extensões.

Este módulo foi inicialmente desenvolvido em C++ e, depois, teve todo o código migrado para Rust[1]. Um dos motivos foi o fato de que rapidamente se tornou desafiador manter uma grande base de código livre de *bugs* relacionados a ponteiros nulos e liberações duplas (*double frees*). Isto acontece porque muitas operações em C e C++ podem gerar erros que podem ser ignorados, sem que uma ferramenta de análise estática de código

[1] https://www.rust-lang.org/

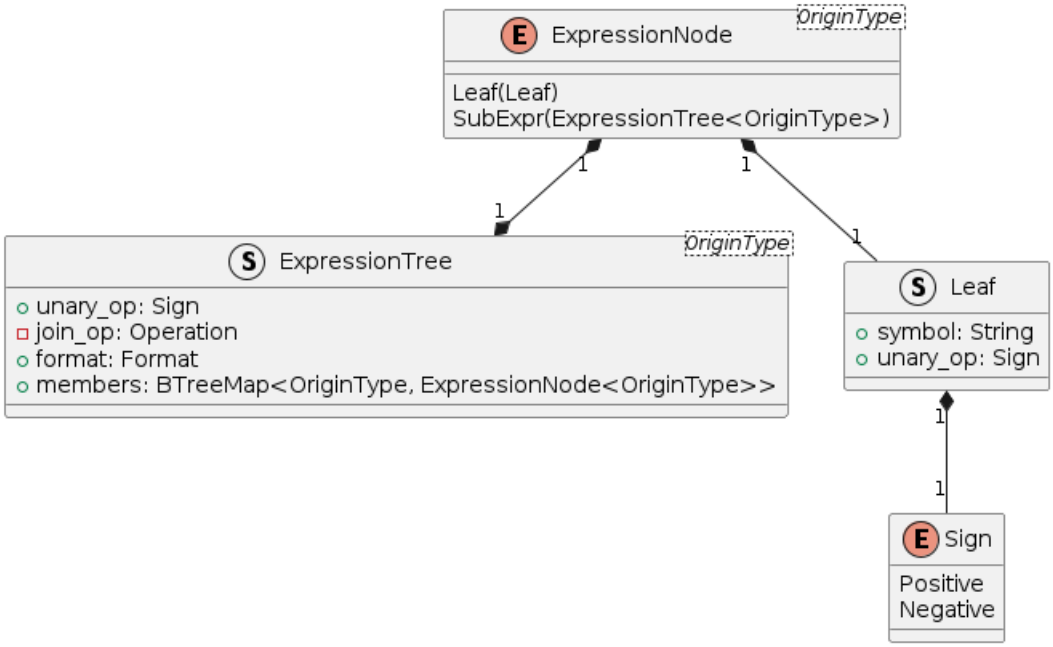

Figura 12 – Diagrama de classes do módulo da árvor de expressões.

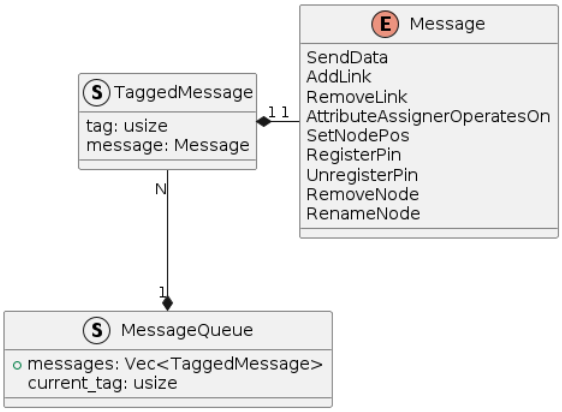

Figura 13 – Diagrama de classes do módulo de mensagens.

consiga auxiliar nestes casos. Ademais, mesmo os recursos mais recentes de segurança de memória adicionados em C++ não podem ser completamente utilizados, uma vez que não possuem uma ABI (*Application Binary Interface*) estável e, portanto, não podem ser utilizados intercambiavelmente entre C++ e Rust.

A importância de evitar *bugs* relacionados à segurança de memória foi ressaltada pela agência de segurança nacional dos EUA em novembro de 2022[2], que, no documento citado, recomendou o uso das chamadas linguagens seguras no uso da memória (*memory safety languages*) como, por exemplo, Rust, C#, Go, Java, Python and Swift. A segurança de memória é uma propriedade de algumas linguagens de programação que evita que os programadores introduzam certos tipos de *bugs* relacionados ao uso da memória como ponteiros pendentes, ponteiros nulos, *buffer overflow*, entre outros. Outro motivo para

[2] Software Memory Safety | U/OO/219936-22

a mudança de linguagem foi a disponibilidade de bibliotecas na linguagem Rust que facilitaram bastante o trabalho de salvar e carregar a RI, e também de gerar código.

Durante o desenvolvimento, tentou-se uma solução que combinava C++ para a interface gráfica e Rust para a geração de código e gerenciamento da representação intermediária. Porém, foram encontradas complicações na implementação da interoperabilidade entre as linguagens. Uma dessas complicações se referiu ao fato de que qualquer mudança na RI impactava consideravelmente o código responsável pela comunicação entre as linguagens. Devido a todas essas questões mencionadas, optou-se por migrar o código para Rust.

4.4.2 A biblioteca de RI

Com o objetivo de desacoplar a aplicação gráfica das conversões realizadas, o módulo da representação intermediária foi desenvolvido como uma biblioteca à parte. Desta forma, as conversões para JSON e código em Python devem ser realizadas por esta biblioteca, e apenas as representações intermediárias válidas podem ser construídas pela aplicação.

Esta abordagem facilita o processo de testagem do módulo. Criaram-se vários testes, unitários e de integração, partindo de representações válidas. Como as funções públicas do módulo só possibilitam a criação de representações válidas, então é garantido que a GUI sempre gera modelos que são armazenáveis em disco, simuláveis e exportáveis.

A Figura 14 ilustra os fluxos de simulação e exportação, e como os módulos da GUI e RI interagem durante este processo.

A geração de JSONs é feita de forma transparente e automática a partir da estrutura de dados que armazena o modelo na RI. A geração de código, por outro lado, é realizada a partir da substituição de templates com os dados das RI.

4.5 Distribuição do software

A distribuição de software é um tema complexo porque não existe uma única maneira correta de ser feita; todas as abordagens trazem seus prós e contras. Contudo, dependendo de como for feita, pode aumentar ou diminuir o alcance do software e seu público-alvo. Por exemplo, desenvolvedores de softwares cujo público-alvo é composto de programadores experientes em uma certa linguagem de programação podem supor que seus usuários tenham as ferramentas necessárias instaladas e, portanto, podem compilar o código fonte ou instalá-lo por algum gerenciador de pacotes. Exemplos de softwares distribuídos dessa maneira são os compiladores das linguagens de programação C, C++ e Rust.

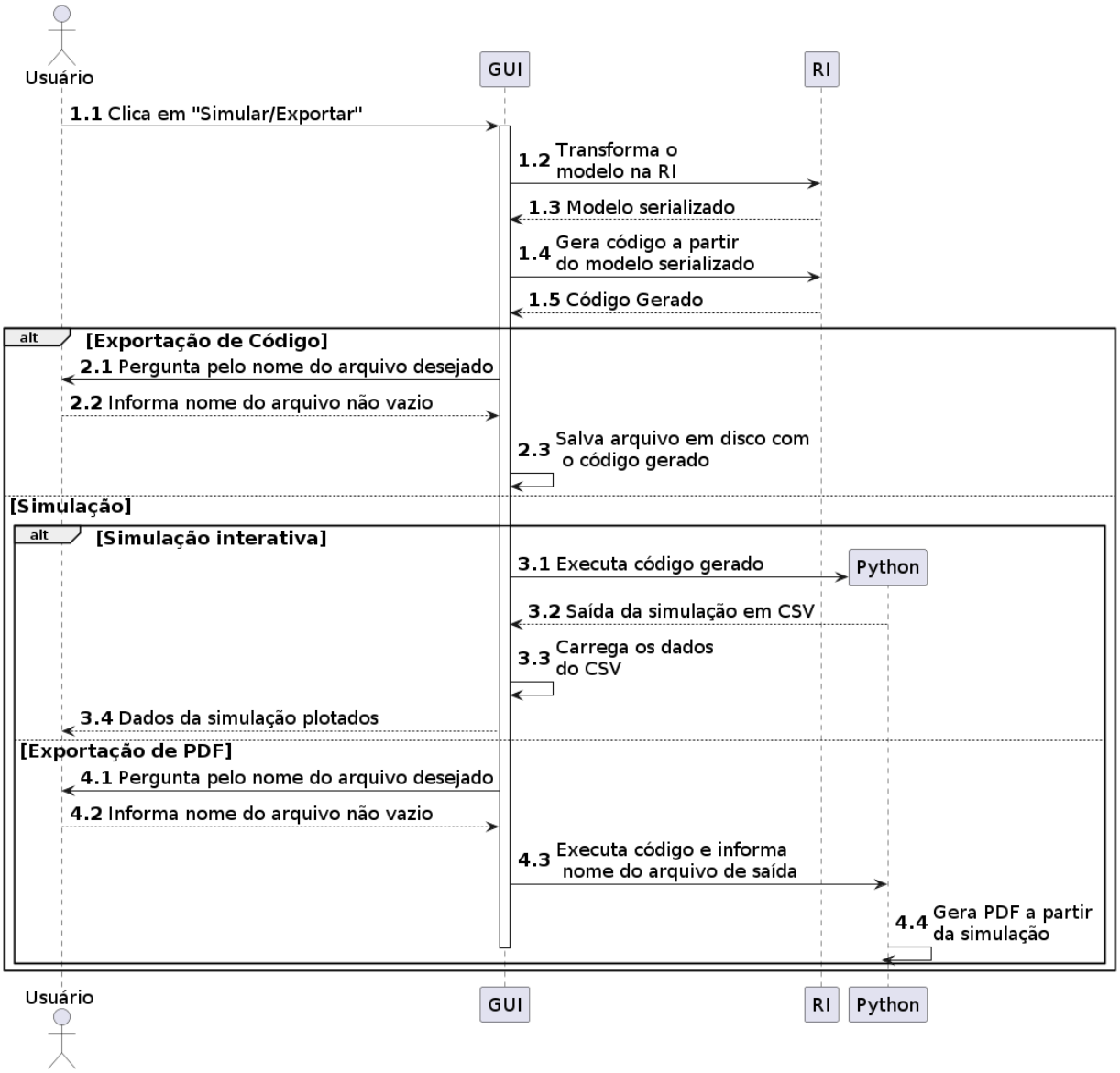

Figura 14 – Diagrama de sequência representando o processo de exportação de código, simulação interativa e exportação de resultados.

Por outro lado, é impensável que desenhistas possuam as dependências necessárias para compilar softwares como o Krita[3]. Por isso, o software disponibiliza instaladores e executáveis portáteis diretamente pelo site para todos os sistemas operacionais suportados, como mostra a Figura 15.

Como os objetivos do software desenvolvido neste trabalho são (1) auxiliar no ensino-aprendizagem de modelagem computacional e (2) auxiliar pesquisadores que não são necessarimante programadores, é imprescindível que o software seja facilmente instalado, preferencialmente sendo distribuído como um executável portátil.

Contudo, como explicado no Capítulo 4, o software depende externamente de um interpretador de Python e a biblioteca `scipy`. Seria necessário que o usuário instalasse es-

[3] Krita Website

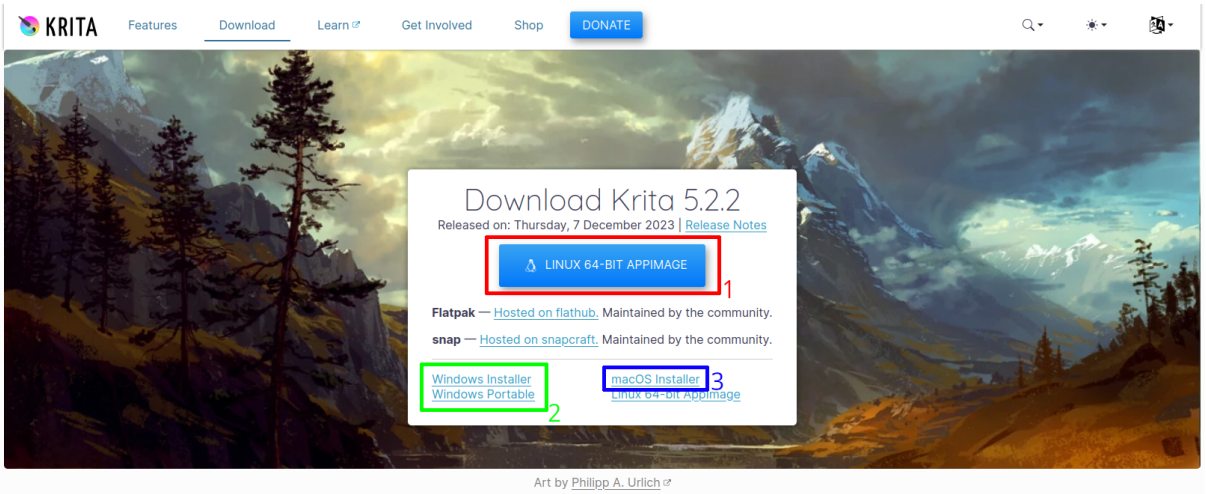

Figura 15 – Página principal do website do software Krita, apresentando: (1) download de executável portátil para Linux, (2) downloads de executável portátil e instalador para Windows e (3) download de instalador para Mac.

tas dependências por conta própria, o que poderia impactar negativamente a sua facilidade de uso. Ademais, instalações manuais de versões incorretas das dependências poderiam afetar o funcionamento do software, o que causaria confusão para o usuário final.

Com isto em mente, foi decidido distribuir o software em conjunto com todas as dependências "vendorizáveis", isto é, dependências que podem ser distribuídas em versões específicas e possivelmente diferentes das encontradas no sistema operacional.

4.5.1 Empacotamento de AppImages para sistemas Linux

Sistemas operacionais baseados em Linux são notavelmente diferentes quanto a seus gerenciadores de pacotes, que muitas vezes utilizam softwares e formatos diferentes. Para um desenvolvedor de software, distribuir um programa nativamente nas mais populares distribuições Linux significa manter múltiplas versões deste, compiladas contra diferentes versões de suas dependências e devidamente testadas. Também é necessário continuamente compilar o software com novas versões, visto que as dependências eventualmente serão atualizadas.

Em contraste com esta abordagem, um formato denominado AppImage[4], inicialmente lançado em 2004, foi criado com o propósito de possibilitar que um pacote seja distribuído para múltiplos sistemas operacionais baseados em Linux sem requerer alterações.

Um arquivo seguindo a especificação do formato inclui múltiplos arquivos armazenados numa estrutura de diretórios similar àquela dos sistemas Linux. Quando executada, a AppImage será descompactada em RAM como um sistema de arquivos virtual, e um ar-

[4] AppImage Website

quivo específico será executado como ponto de entrada do software. A Figura 16 apresenta a hierarquia de diretórios presente na AppImage do software desenvolvido.

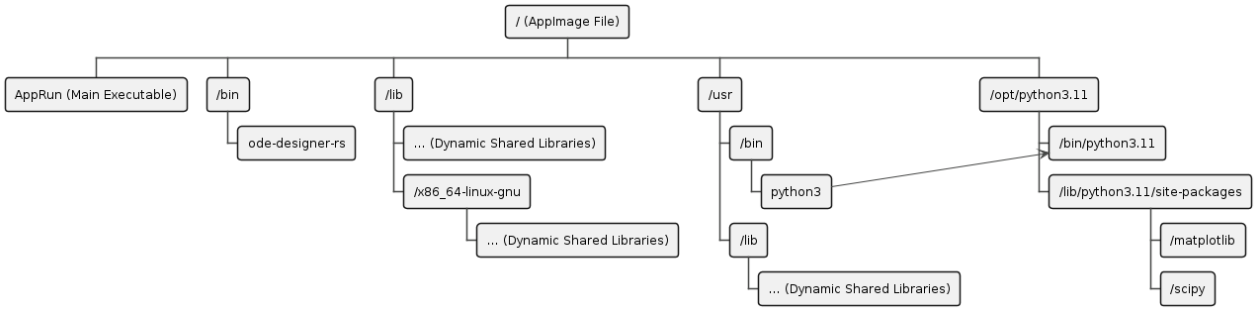

Figura 16 – Estrutura de diretórios da AppImage do software.

A AppImage contém a maioria das dependências de bibliotecas dinâmicas e inclui o interpretador de Python com as dependências necessárias pré-instaladas. Exceções às bibliotecas dinâmicas inclusas são as relacionadas às bibliotecas padrões de C e C++, que não devem ser distribuídas em versões mais novas. Como estas bibliotecas mantém retro-compatibilidade em novas versões, o software deve ser compilado numa distribuição com a menor versão suportada, garantindo funcionamento com versões mais novas.

Com o intuito de realizar este processo automaticamente, uma esteira de lançamento de versões foi desenvolvida. Ela se encarrega de compilar o software em um contêiner de Docker (MERKEL, 2014) com uma versão de 2021 do sistema Debian com todas as dependências necessárias para fazer o software funcionar em sistemas Linux e Windows.

4.5.2 Empacotamento ZIP para sistema Windows

Ao contrário dos sistemas Linux, não existe um formato de arquivos que contenha múltiplos arquivos internamente e seja executável ao mesmo tempo no sistema Windows. Por este motivo, softwares para o SO Windows são comumente distribuídos como pastas compactadas (ZIP) ou instaladores que populam uma pasta com os arquivos necessários.

Portanto, o software é distribuído para Windows por meio de uma pasta compactada incluindo as dependências dinâmicas e um interpretador de Python portátil, que inclui o `scipy`.

4.6 Utilizando o software

Nesta Seção, o software desenvolvido será utilizado para representar o modelo de difusão de Bass (BASS, 1969), que descreve o processo de adoção de novos produtos por uma população. O modelo de Bass nasce de uma hipótese comportamental na qual se assume que, durante o processo de difusão do uso de um novo produto, dois tipos

de consumidores irão determinar a forma como a demanda irá crescer. O primeiro tipo corresponde àqueles indivíduos que decidem adotar o produto de forma independentemente, ou seja, não recebem influência direta de outros consumidores, todavia, podem ser motivados por outros meios, por exemplo, por comunicação de massa. Esses indivíduos são denominados "inovadores". O segundo tipo corresponde aos consumidores potenciais que são influenciados pela pressão social do meio e que são susceptíveis à influência de outros consumidores que já adquiriram o produto. Estes consumidores são denominados "imitadores".

A EDO, presente na Equação 4.1, modela as interações entre o processo de adoção e influência, classificando os adotantes em inovadores e imitadores.

$$\frac{dF}{dt} = (1 - F)(p + qF) \tag{4.1}$$

Em que,

F	Fração da base de instalação
p	Coeficiente de inovação
q	Coeficiente de imitação

Ao iniciar o software, o usuário se deparará com um editor vazio e um menu lateral com algumas opções de nós para serem criados. A criação dos nós pode ser feita tanto pelo menu lateral (destacado como '1' na Figura 17), quanto pelo menu de contexto acessível pelo botão direito do mouse (destacado como '2' na Figura 17).

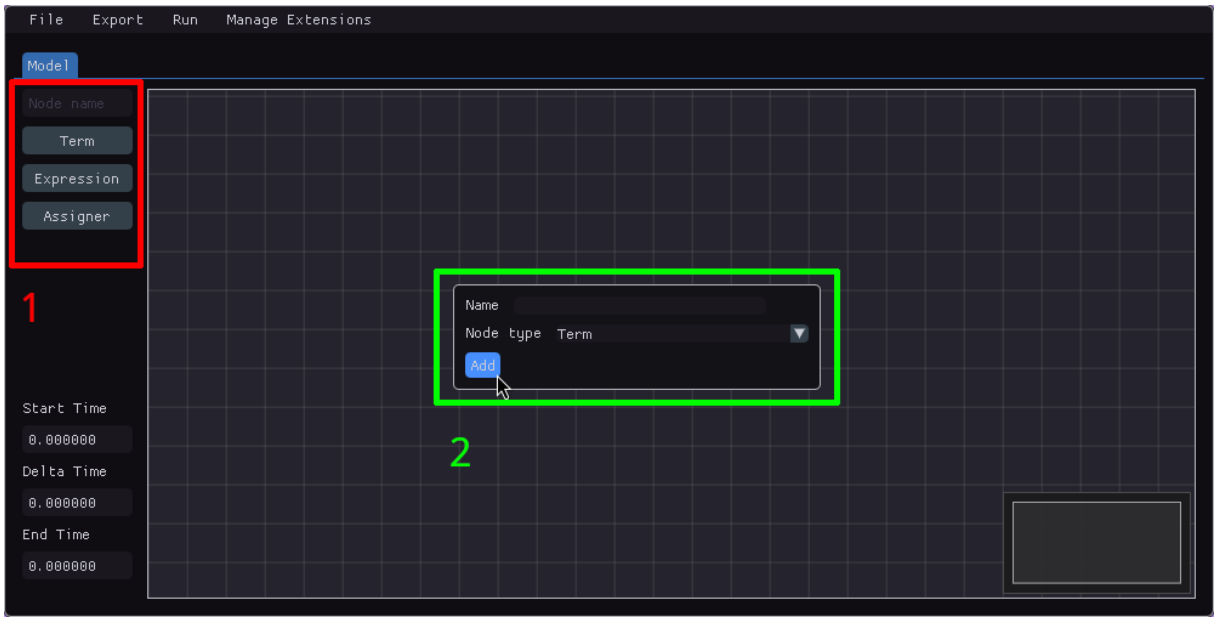

Figura 17 – As opções de criação de nós. Destacado como '1' está o menu lateral com as opções de nós e o campo de nome. Destacado como '2' está o menu de contexto acessível com o clique do botão direito do mouse.

No software, é necessário criar um nó de termo para cada variável, parâmetro e constante. Portanto, pode-se afirmar que a Equação 4.1 possui os seguintes termos (presentes na Figura 18):

- 1 variável: F;
- 2 parâmetros: p e q;
- 1 constante: o número 1.

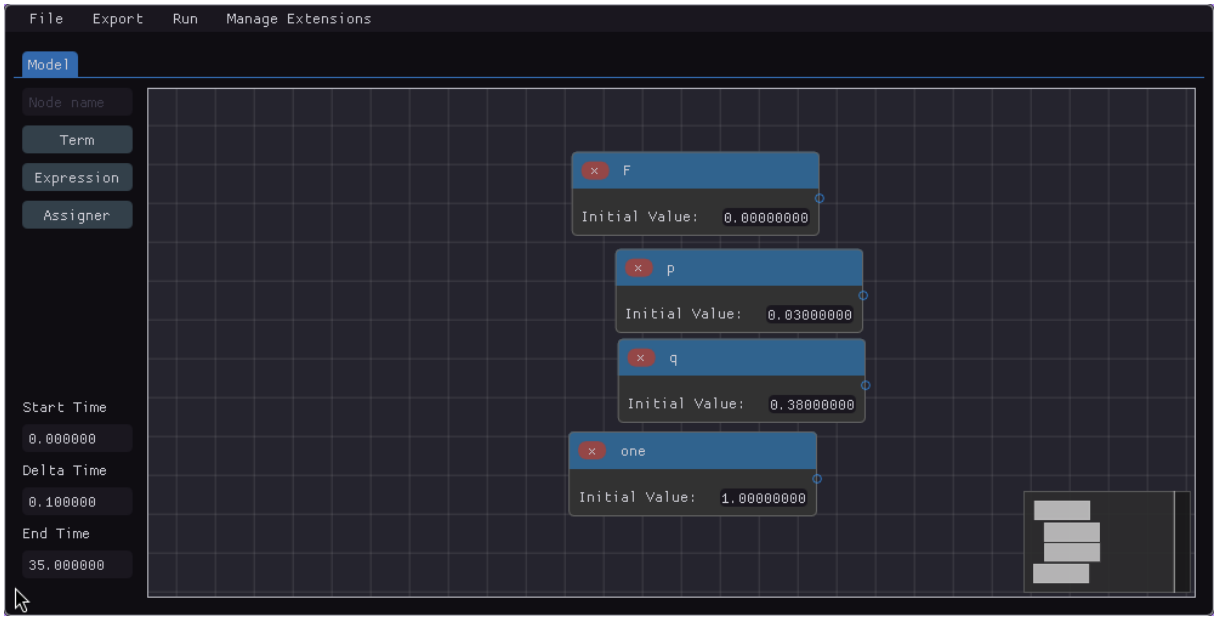

Figura 18 – Os nós termos criados. Note que a constante "one" (em inglês "um") tem seu valor definido como 1. Os outros termos possuem os valores $F = 0$, $p = 0.03$ e $q = 0.38$.

A próxima etapa da construção do modelo envolve as expressões. No software, cada expressão pode ser usada para combinar duas ou mais subexpressões (incluindo termos) utilizando apenas uma operação matemática. Desta forma, é necessário um nó de expressão para cada operação binária. A Figura 19 apresenta as subexpressões e termos necessários para a construção da Equação 4.1, apresentando as 4 subexpressões necessárias que são demonstradas no software pela Figura 20.

A última etapa na construção de um modelo envolve atribuir equações a variáveis. No caso da Equação 4.1, há apenas uma Equação, $\frac{dF}{dt} = \ldots$. Na Figura 21 é demonstrado um nó para esta Equação, faltando apenas atribuí-la a uma variável. A atribuição da variável garante que um termo terá seu valor atualizado entre diferentes tempos da simulação.

Finalmente, as Figuras 22 e 23 apresentam, respectivamente, o modelo final e o resultado da simulação.

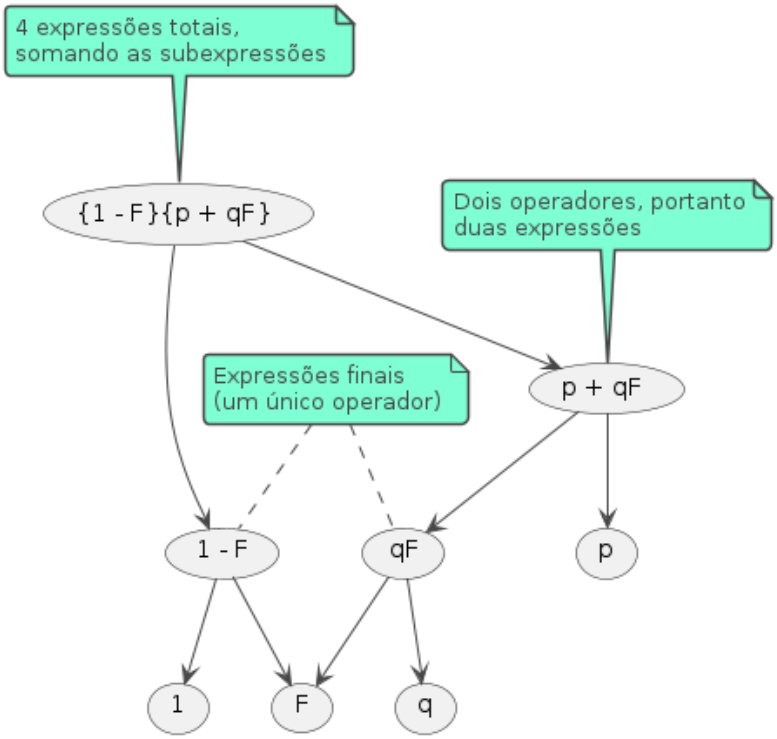

Figura 19 – As subexpressões e termos usados para construir a Equação 4.1. Esta desconstrução é útil para a representação no software.

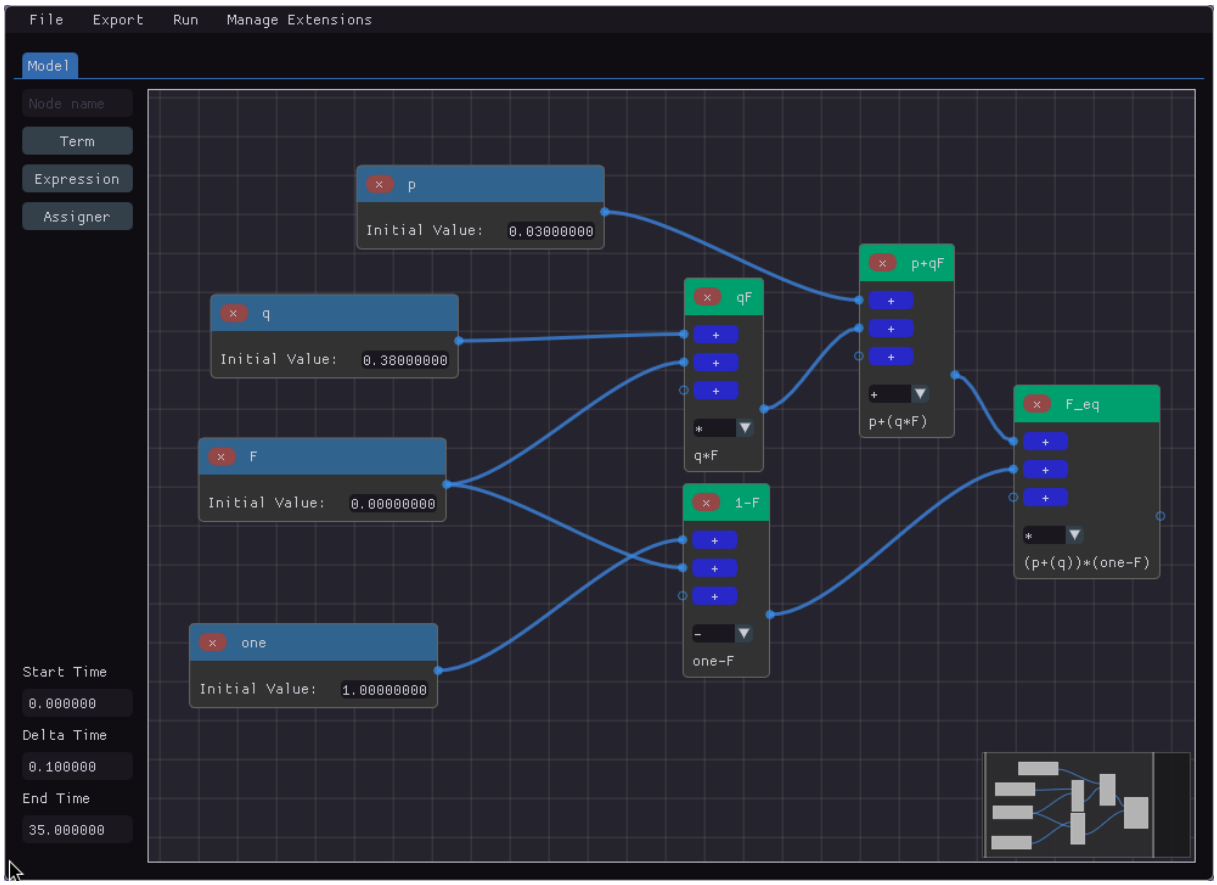

Figura 20 – As 4 expressões construídas no software: (1) qF, (2) $1-F$, (3) $p+qF$ e (4) $(p+qF)*(1-F)$.

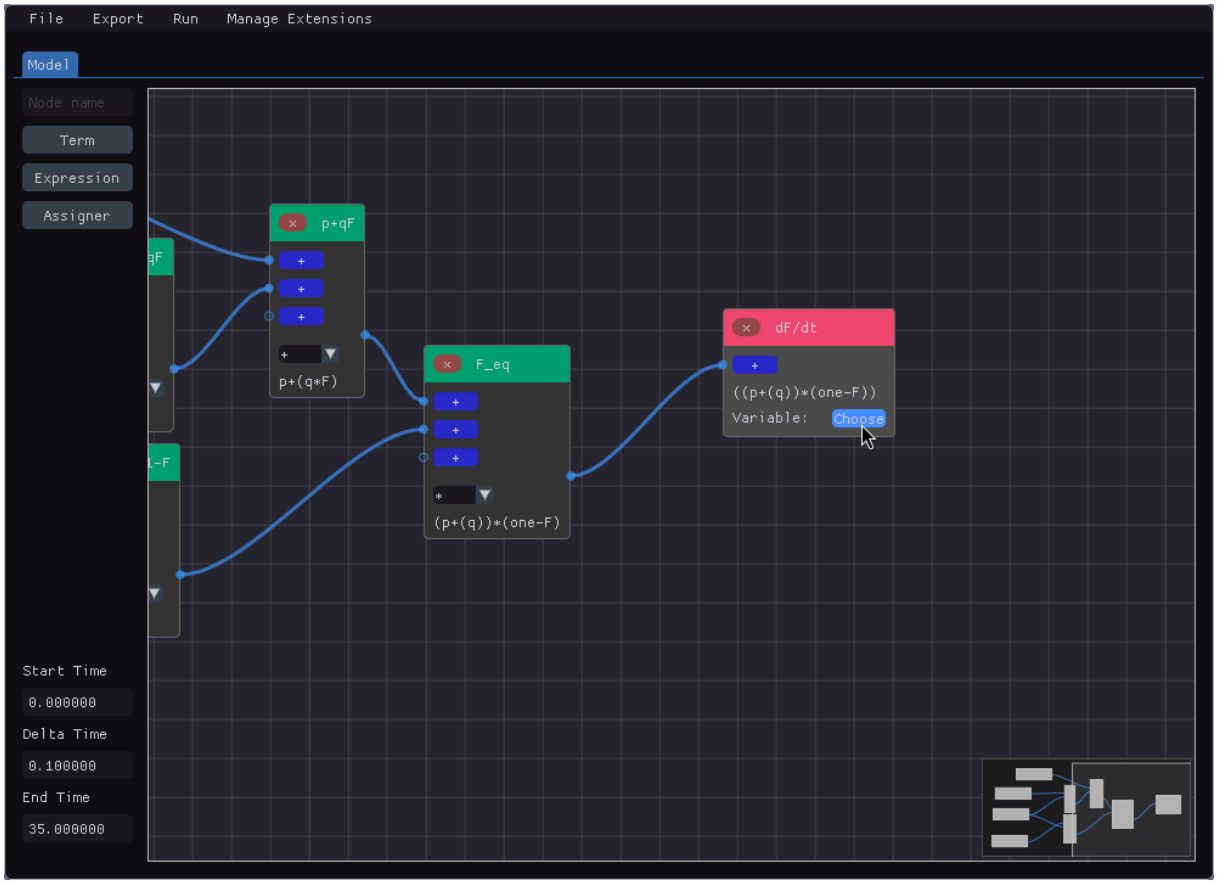

Figura 21 – A Equação $\frac{dF}{dt} = \ldots$ descrita como um nó. Nesta etapa, é necessário selecionar um termo como variável, clicando no botão "choose" (em inglês, "escolher").

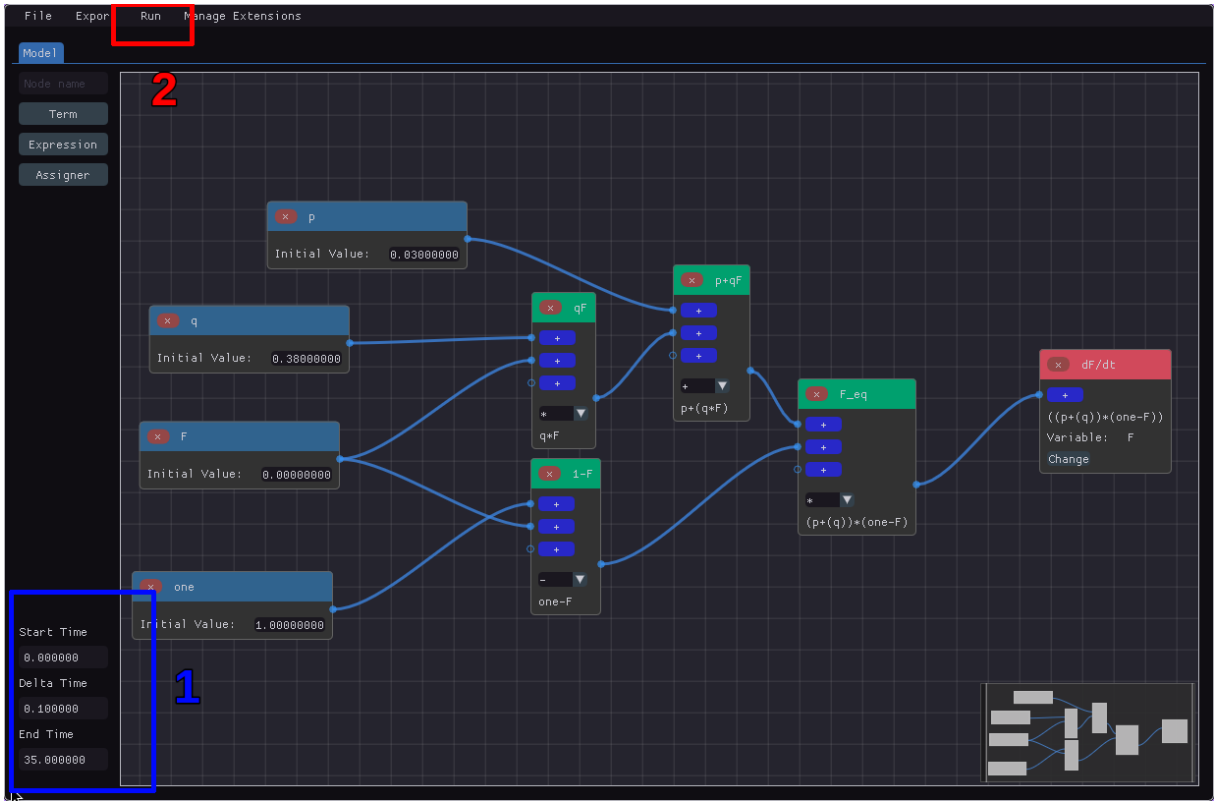

Figura 22 – O modelo final construído para a Equação 4.1. Destacado como '1' estão os itens que controlam os tempos das simulações, que foram alterados para 0.0 (tempo inicial), 0.1 (tempo delta) e 35.0 (tempo final). Descatado como '2' está o botão para realizar a execução e simulação do programa.

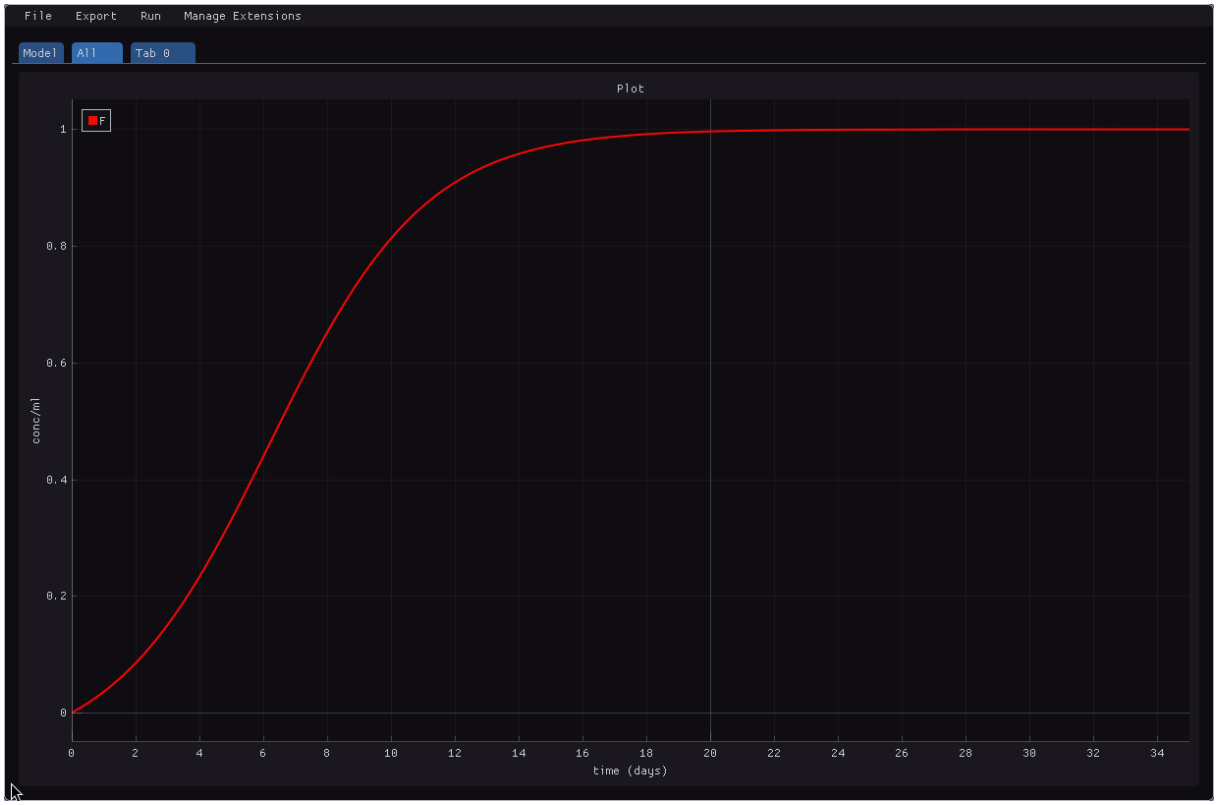

Figura 23 – O resultado da simulação para os parâmetros $F_0 = 0$, $p = 0.03$, $q = 0.38$.

5 Aplicação do software na modelagem computacional

Nas seções a seguir, são apresentados alguns exemplos de modelos construídos e simulados com o software ODE-Designer.

5.1 Predador-presa

O modelo predador-presa foi descrito na Seção 2.1. A Figura 24 mostra o modelo construído no software e a Figura 25 mostra um exemplo de resultado da simulação.

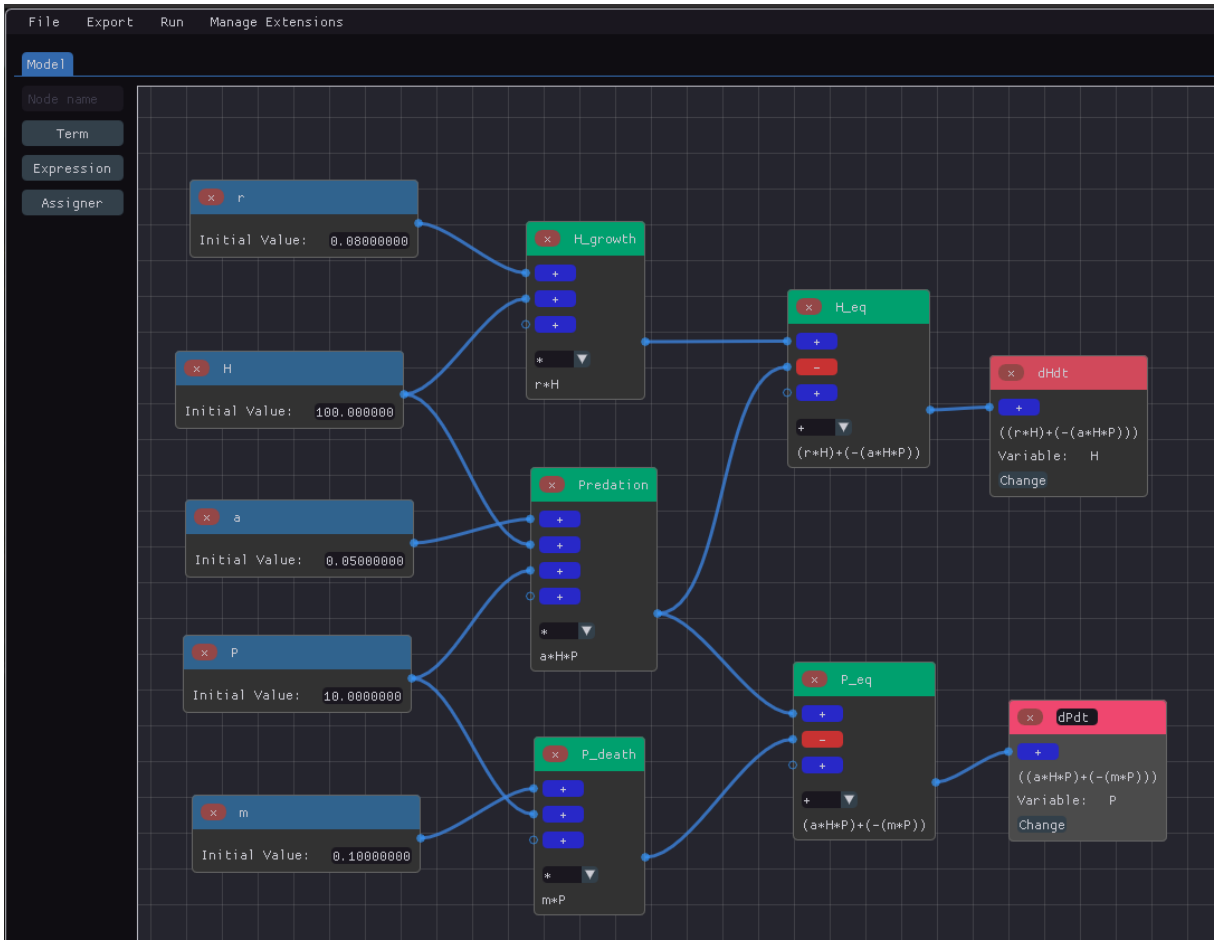

Figura 24 – Modelo predador-presa construído no software.

5.2 Transmissão de doenças

Existem alguns modelos de EDOs clássicos para modelar a transmissão de um vírus na população como, por exemplo, os modelos SIR, SIRS e SIS. Para mostrar a aplicação

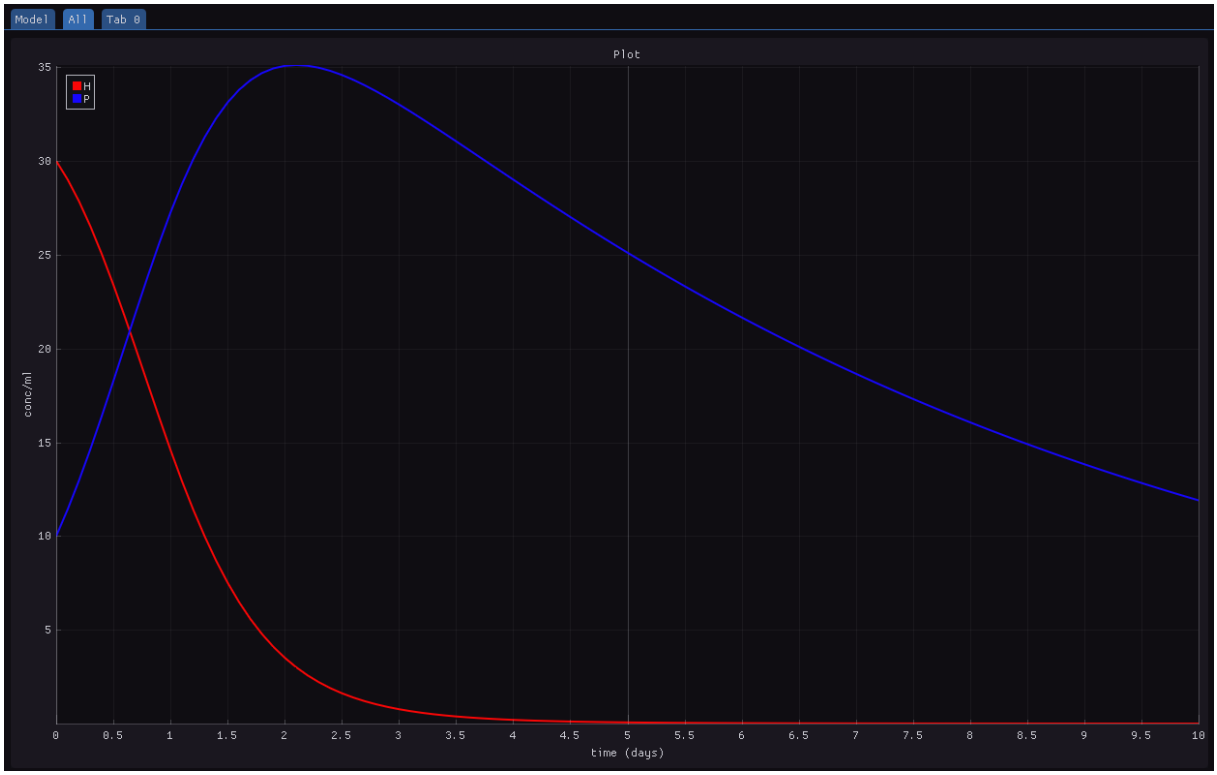

Figura 25 – Exemplo de resultado da simulação do modelo predador-presa.

do software, foi construído o modelo SIRS que é composto pelo sistema de EDOs dado na Equação 5.1.

$$\begin{aligned} \frac{dS}{dt} &= -\beta.S.I + \gamma.R \\ \frac{dI}{dt} &= \beta.S.I - \alpha.I \\ \frac{dR}{dt} &= \alpha.I - \gamma.R \end{aligned} \tag{5.1}$$

O modelo SIRS divide a população em indivíduos suscetíveis (S) à doença, infectados (I) e recuperados (R). Uma parte da população de suscetíveis pode se infectar (termo $\beta.S.I$). Os indivíduos infectados podem se recuperar com uma determinada probabilidade (termo $\alpha.I$) e os indivíduos recuperados podem voltar a ser suscetíveis à doença (termo $\gamma.R$).

Na Figura 26, podemos ver o modelo SIRS construído no software e, na Figura 27, um exemplo de resultado da simulação do modelo.

5.3 Reação enzimática

O modelo da reação enzimática com inibição competitiva modela uma reação química entre um substrato S e uma enzima E para formar um produto P. A enzima atua

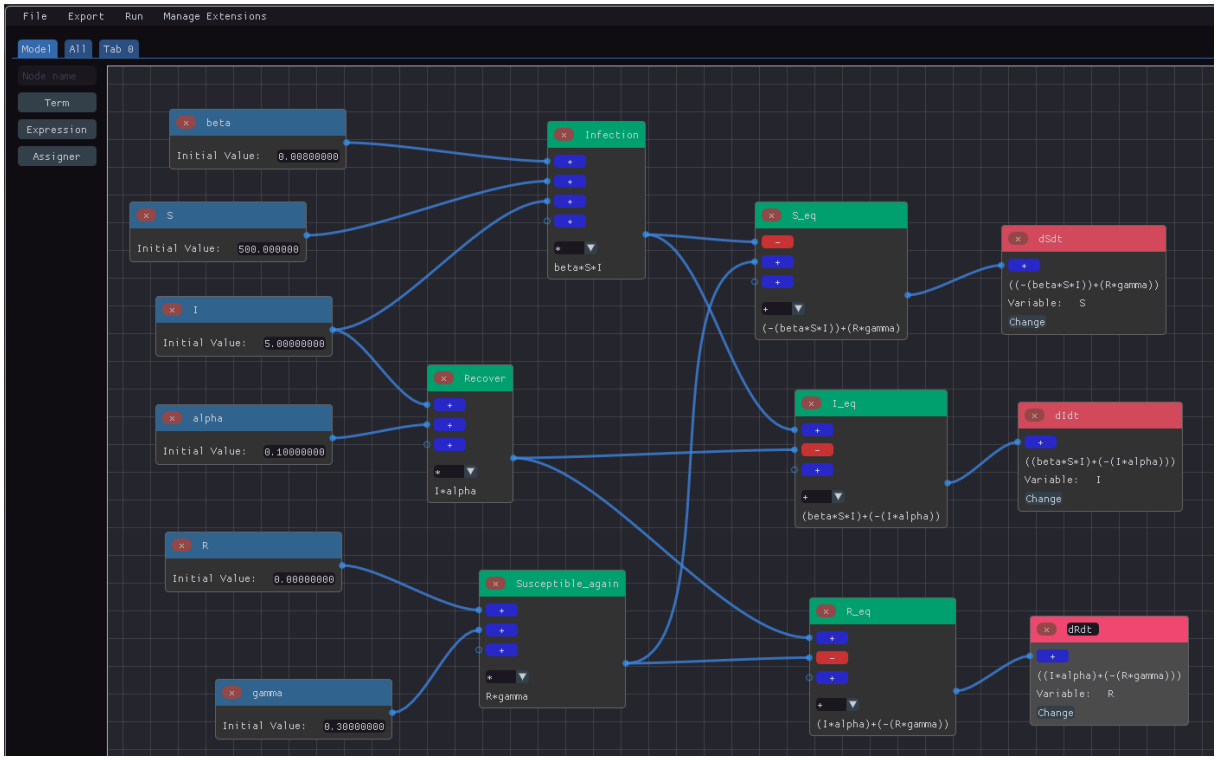

Figura 26 – Modelo SIRS construído no software.

catalisando a reação e formando um complexo temporário ES que pode voltar a se transformar em $E + S$ ou virar $E + P$. Essa reação é inibida por uma substância I que se liga ao mesmo sítio de ligação do substrato S na enzima E formando um outro complexo chamado EI. Essa ligação impede a ligação de S com E. O complexo EI também tem um tempo de vida finito e pode voltar a ser $E + I$. O sistema de EDOs que modela essa reação é dado na Equação 5.2.

$$
\begin{aligned}
\frac{dS}{dt} &= k_2.ES - k_1.S.E \\
\frac{dP}{dt} &= k_3.ES \\
\frac{dI}{dt} &= k_5.EI - k_4.E.I \\
\frac{dES}{dt} &= -k_2.ES + k_1.S.E - k_3.ES \\
\frac{dEI}{dt} &= -k_5.EI + k_4.E.I \\
\frac{dE}{dt} &= k_2.ES + k_3.ES + k_5.EI - k_1.S.E - k_4.E.I
\end{aligned}
\tag{5.2}
$$

A Figura 28 mostra o modelo da reação enzimática construído no software e a Figura 29 mostra um exemplo de resultado do modelo.

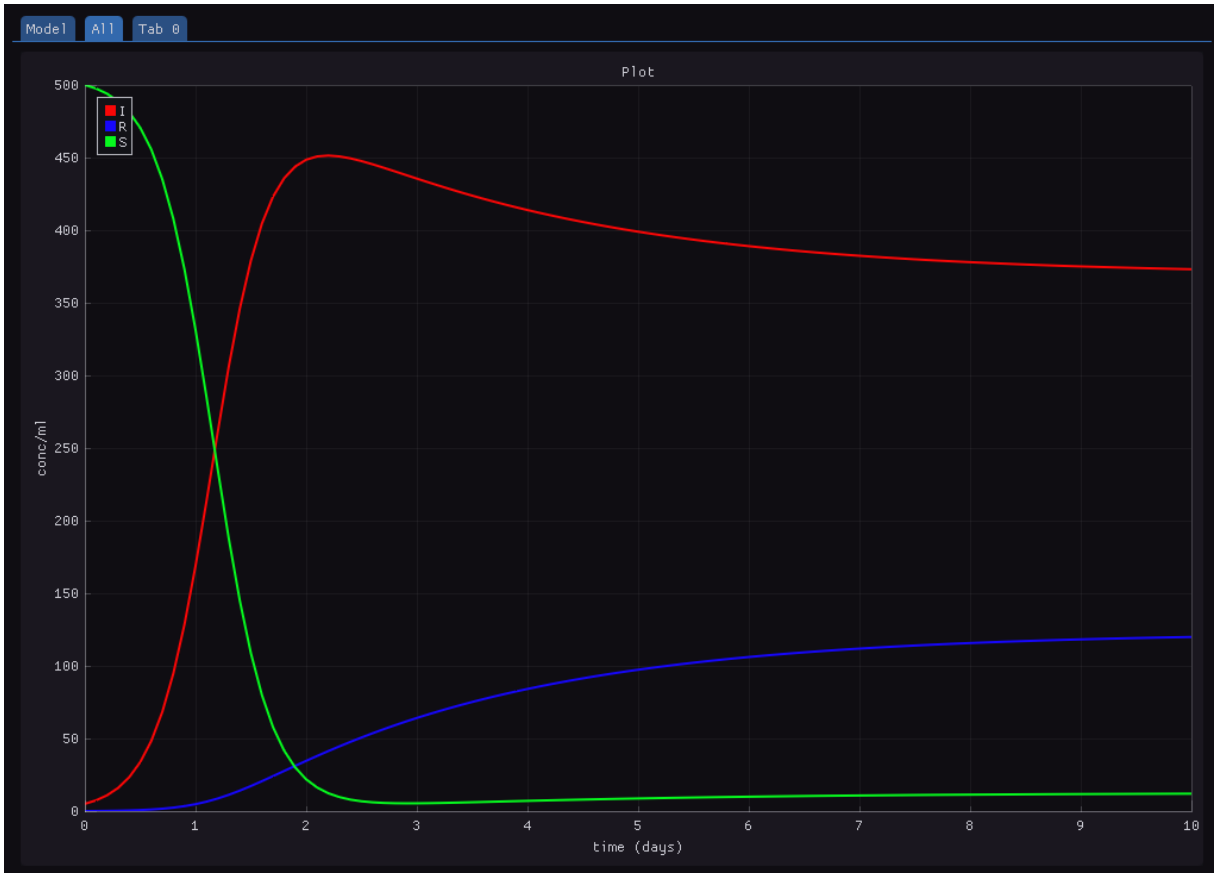

Figura 27 – Exemplo de resultado da simulação do modelo SIRS.

5.4 Resposta imune a uma infecção viral

O modelo apresentado nesta Seção modela a resposta imune das células T a uma infecção viral. O modelo é descrito a seguir.

As células epiteliais saudáveis (E) são infectadas pelo vírus (V) tornando-se infectadas (I). As células T efetoras também causam dano ao tecido e esse dano é quantificado através de um marcador de inflamação chamado D. As células infectadas são eliminadas pelas células T ou podem morrer devido à ação dos vírus intracelulares. Quando a célula infectada morre devido à ação dos vírus, são liberados novos vírus no tecido. Os vírus são eliminados pela ação dos anticorpos e outras células de defesa que não são explicitamente representadas no modelo. As células T migram para o tecido devido à ação das citocinas pró-inflamatórias (C). Esse processo foi simplificado porque não considera-se, no modelo atual, as células T *helper* e outros níveis de ativação das células T de forma explícita. As citocinas pró-inflamatórias são produzidas pelas células T na presença de marcadores inflamatórios da infecção.

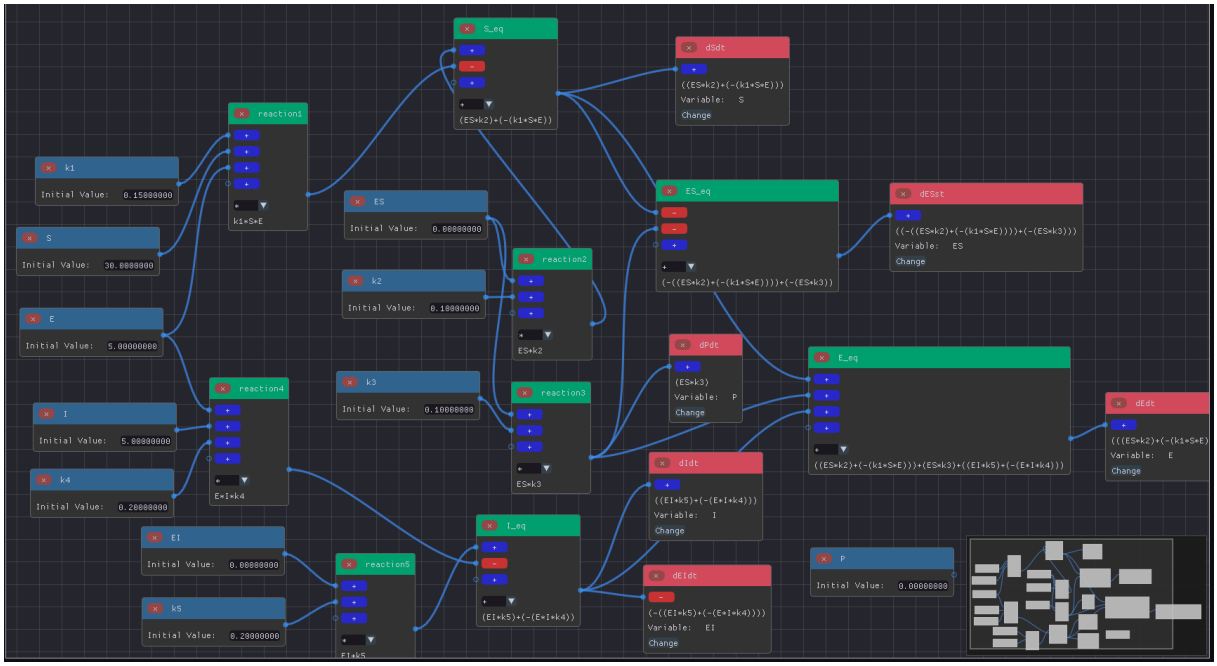

Figura 28 – Modelo da reação enzimática construído no software.

$$
\begin{aligned}
\frac{dD}{dt} &= \beta.(T + 2.I) - m_D.D \\
\frac{dE}{dt} &= -k_E.E.V \\
\frac{dI}{dt} &= k_E.E.V - k_I.I.T - a.I \\
\frac{dV}{dt} &= a.I - k_V.V \\
\frac{dT}{dt} &= s_T.(1 + C) - m_T.T \\
\frac{dC}{dt} &= beta_C.T.(I + V) - m_C.C
\end{aligned}
\tag{5.3}
$$

Na Figura 30, é apresentada parte do modelo construído no software.

Um exemplo de resultado do modelo, é apresentado nas Figuras 31 e 32.

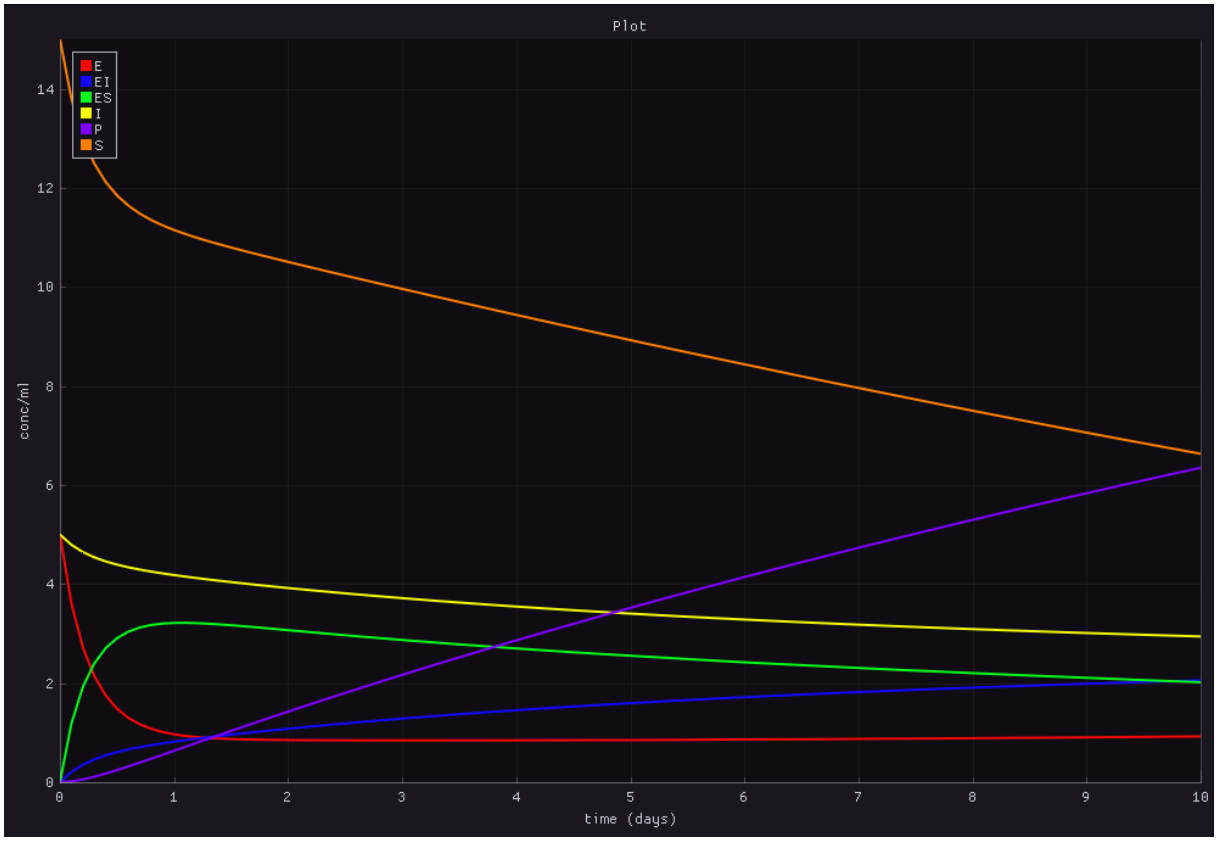

Figura 29 – Exemplo de resultado do modelo da reação enzimática.

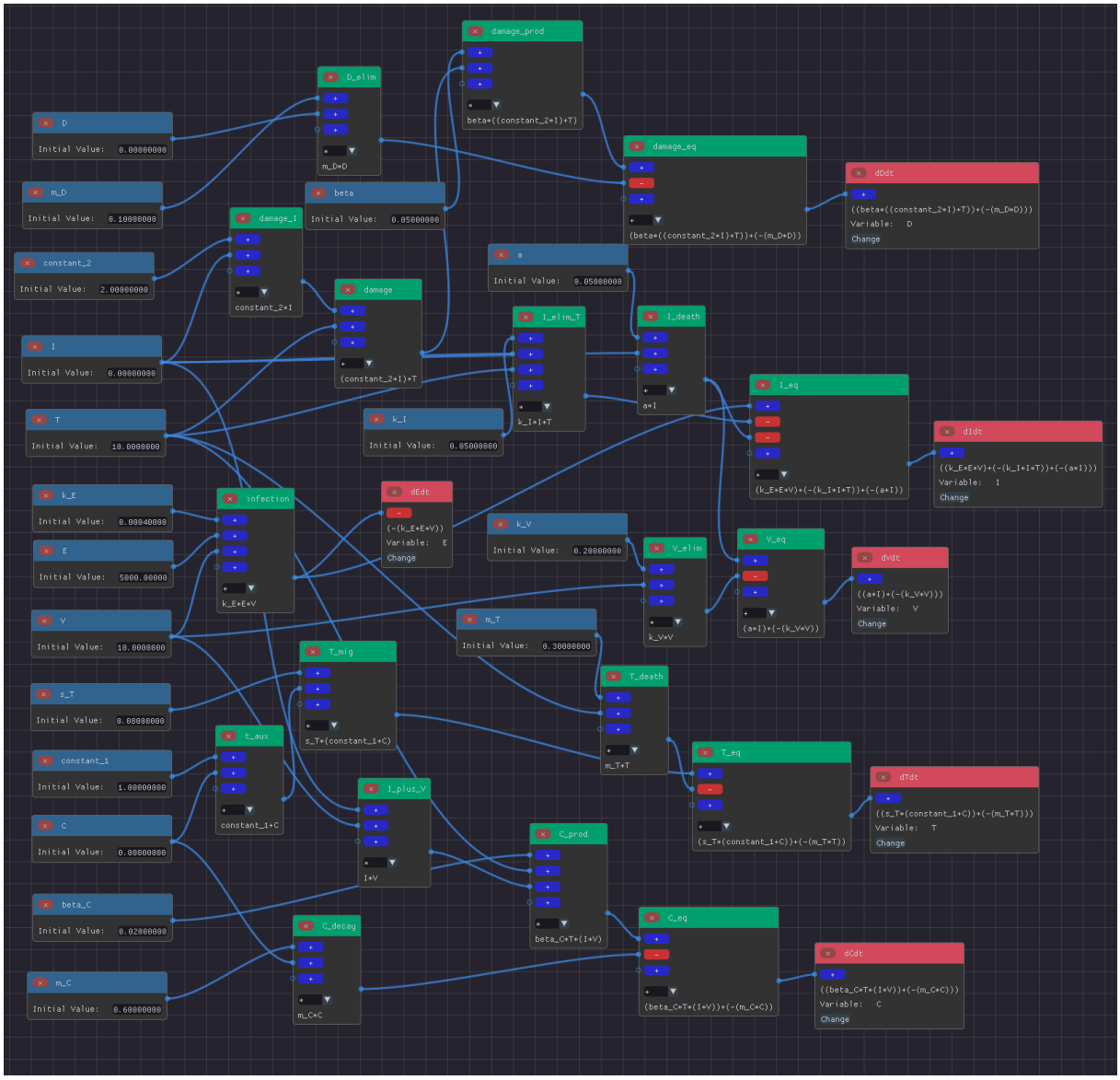

Figura 30 – Modelo da infecção viral construído no software.

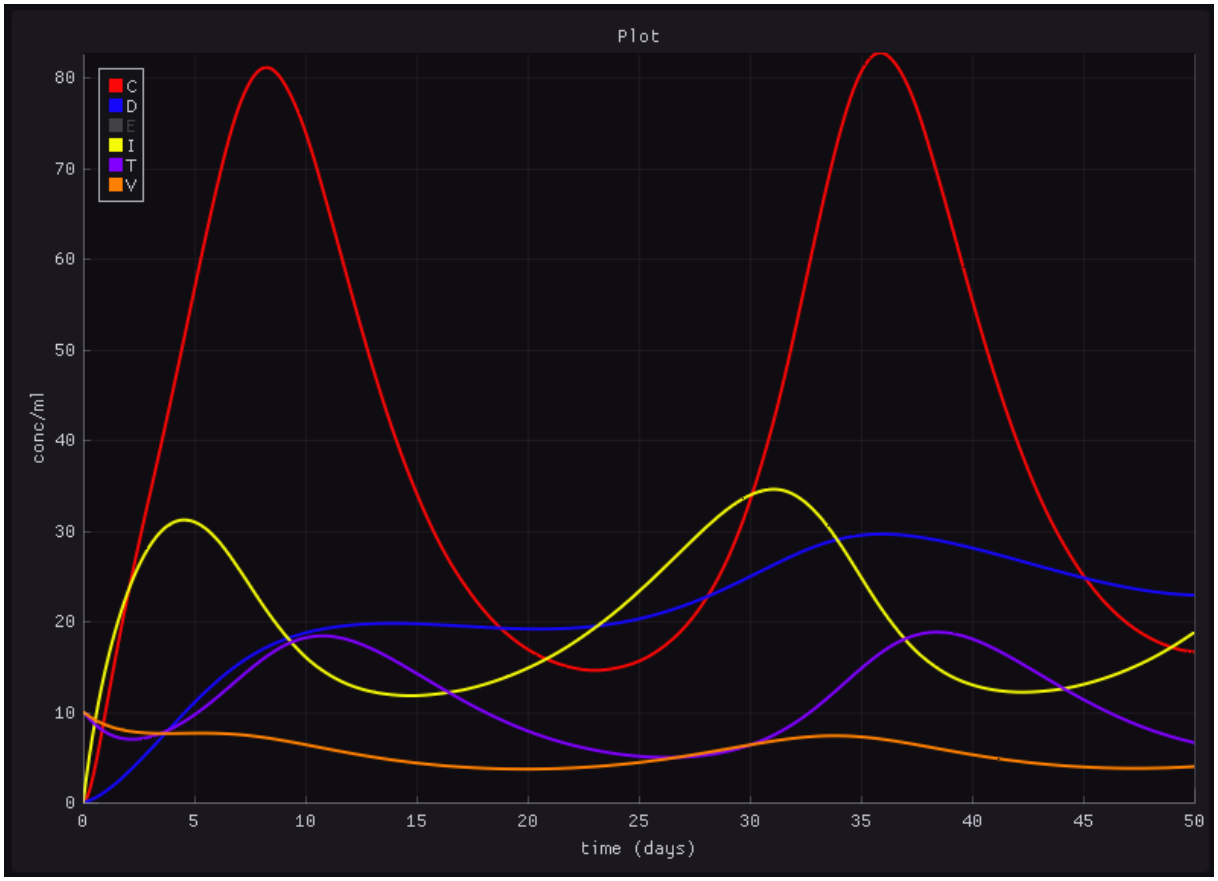

Figura 31 – Exemplo de resultado do modelo da infecção viral.

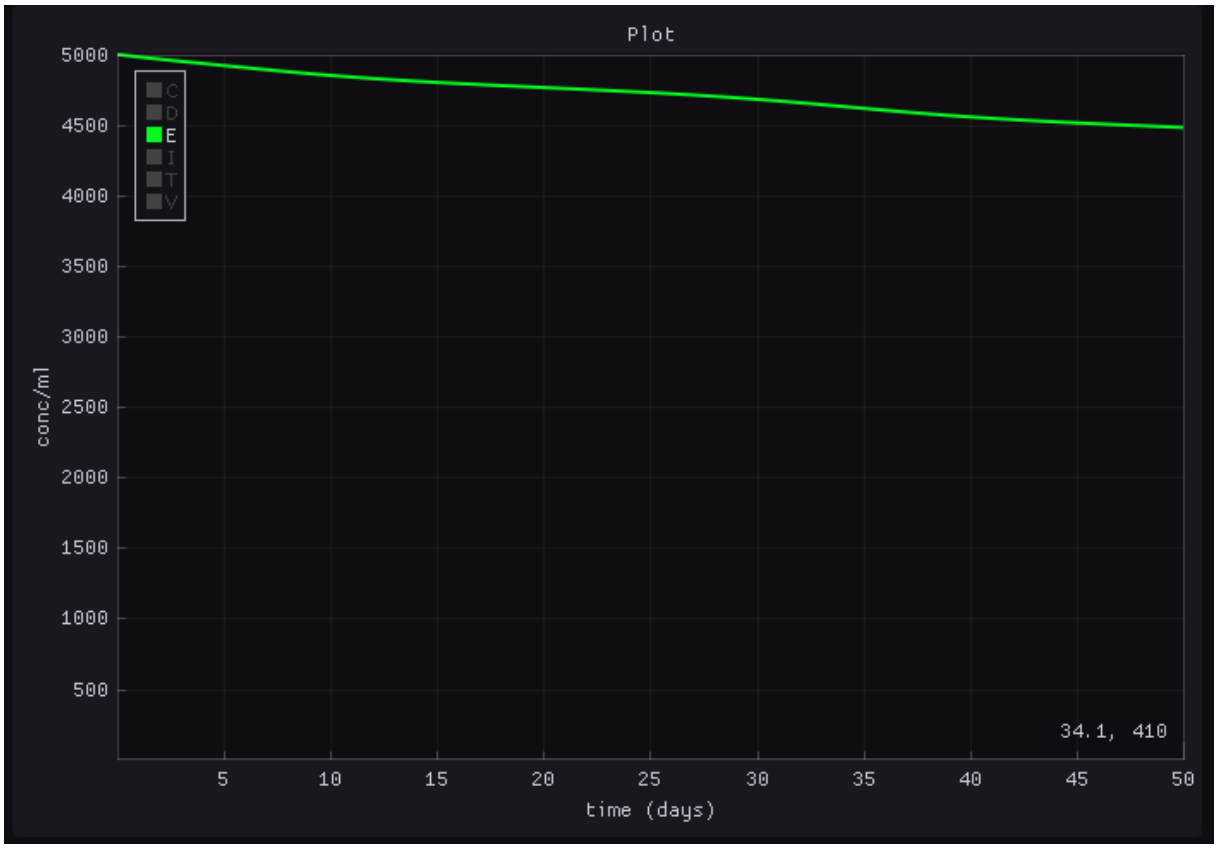

Figura 32 – Exemplo de resultado do modelo da infecção viral mostrando a variável E.

6 Conclusão e trabalhos futuros

Neste trabalho, foi desenvolvido um software para automatizar a implementação e simulação de modelos computacionais baseados em EDOs. Por meio do uso da interface gráfica do software, o usuário pode construir e simular um modelo sem desenvolver uma linha de código. Isto confirma a hipótese levantada pelo trabalho de que é possível automatizar certas etapas do processo de modelagem. Além disso, a construção das equações do modelo matemático, uma etapa fundamental e ao mesmo tempo desafiadora, é auxiliada pela representação visual que foi criada, visto que esta permite que o usuário acompanhe a construção de todas as expressões matemáticas do modelo e como elas estão sendo combinadas para formar o sistema de EDOs. É possível verificar de forma direta, na interface, as entradas e operações de cada expressão, os sinais de cada entrada, de quais expressões um termo participa e quais expressões fazem parte de uma determinada EDO, entre outras informações relevantes para a construção correta do modelo.

A partir da construção de modelos no software, foi possível verificar que ele possui recursos suficientes para a construção de modelos mais simples até modelos mais complexos. Acredita-se que o software poderá ser utilizado efetivamente no processo de ensino-aprendizagem de modelagem computacional e também na pesquisa por profissionais de diferentes áreas por ser uma ferramenta que facilita o desenvolvimento de modelos de complexidade baixa ou média. Acreditamos também que o software pode ser usado como uma ferramenta de prototipação e teste rápido de modelos por automatizar as etapas de implementação e simulação computacional.

A verificação da corretude da implementação gerada pelo software pode ser realizada de algumas formas. Como o usuário tem acesso ao código gerado pelo software, ele pode comparar os resultados do código com os resultados de outra implementação para o mesmo modelo. Por exemplo, ele pode comparar os resultados com alguma implementação considerada de referência para o modelo em questão. Para modelos cuja solução analítica é conhecida, o usuário pode comparar os resultados do código gerado com a solução analítica.

O software atual pode ser estendido de várias formas diferentes como, por exemplo: 1) Com a geração de código em Rust e simulação das EDOs em "tempo real" com a plotagem dos resultados à medida que a resolução do modelo avança no tempo; 2) A introdução da simulação de modelos estocásticos com o algoritmo de Gillespie utilizando o mesmo editor baseado em nós e criando um novo template para geração de código do modelo estocástico; 3) Adição de suporte ao ajuste de parâmetros fornecendo recursos para carregamento dos dados experimentais, escolha das variáveis e parâmetros a serem

ajustados e plotagem dos gráficos e dos dados experimentais após a execução do ajuste; 4) Adição de métodos e *plots* para a análise de sensibilidade de parâmetros; 5) Criação e implementação de uma representação baseada em grafos para uma visualização mais alto nível das interações do modelo. O foco será na visualização de modelos do sistema imune; 6) Adição de novos nós como, por exemplo, nós para difusão, quimiotaxia, com o objetivo de dar suporte a criação de modelos de Equações Diferenciais Parciais (EDPs), e a implementação da geração de código para as EDPs utilizando o método de diferenças finitas; 7) Desenvolvimento de uma versão Web do software para que ele possa ser utilizado por um número maior de pessoas. A criação do modelo será feita através do navegador Web e todo o processamento incluindo a geração de código, execução, geração dos resultados e imagens será feita no servidor. As tecnologias usadas pelo projeto até então são todas compatíveis com a Web. A linguagem Rust, baseada na LLVM, possui suporte nativo à WebAssembly.

O software também pode ser estendido para suportar a importação e exportação de dados em outros formatos de arquivos frequentemente usados na área de biologia computacional como, por exemplo, o formato CellML (CUELLAR et al., 2003).

Atualmente, a representação visual apresenta uma limitação na qual pode se tornar mais difícil entender um modelo dado uma grande quantidade de nós e de ligações. Isto acontece porque muitas ligações se cruzam na interface, e tornam a visualização mais confusa, prejudicando o entendimento do modelo. Uma primeira solução é utilizar os nós de extensão para definir operações mais complexas e criar modelos mais complexos com o uso dos nós definidos reduzindo assim o número de nós de expressão necessários para construir o modelo. Uma segunda solução, relacionada à primeira, seria permitir ao usuário criar um novo nó customizado a partir da combinação de duas ou mais expressões. Por exemplo, suponha que o usuário esteja construindo um modelo que usa expressões no formato $k.(A - B)$, no qual k é um parâmetro e A e B são populações. Atualmente, o usuário precisaria criar dois nós do tipo expressão: um para a operação $(A - B)$ e outro para multiplicar k pelo resultado desta expressão. Na possível solução proposta, o usuário selecionaria os dois nós de expressão e solicitaria ao software para criar um novo nó customizado que receberá três entradas e realizará a operação resultado da combinação das expressões. Uma possível implementação desta solução aproveitaria a extensibilidade do software.

Outra limitação do trabalho foi a não realização de uma avaliação de usabilidade do software. O objetivo seria avaliar aspectos da interface gráfica e da usabilidade no geral, porém não houve tempo hábil para realizar essa avaliação. Planeja-se, futuramente, realizar essa análise de usabilidade em disciplinas da graduação e da pós-graduação que possuam alguma interseção com a área de modelagem computacional.

Referências

ADOMIAN, G. Solving the mathematical models of neurosciences and medicine. *Mathematics and Computers in Simulation*, v. 40, n. 1, p. 107–114, 1995. ISSN 0378-4754. Disponível em: <https://www.sciencedirect.com/science/article/pii/0378475495000218>.

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. Hardcover. ISBN 0321486811. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0321486811>.

AKCAY, B. Problem-based learning in science education. *Journal of Turkish Science Education*, v. 6, 04 2009.

BARELLI, E. Imagining the school of the future through computational simulations: Scenarios' sustainability and agency as keywords. *Frontiers in Education*, v. 7, 2022. ISSN 2504-284X. Disponível em: <https://www.frontiersin.org/articles/10.3389/feduc.2022.897582>.

BASS, F. M. A new product growth for model consumer durables. *Management Science*, Institute for Operations Research and the Management Sciences (INFORMS), v. 15, n. 5, p. 215–227, jan. 1969. ISSN 1526-5501. Disponível em: <http://dx.doi.org/10.1287/mnsc.15.5.215>.

BORRELLI, A.; WELLMANN, J. Computer simulations then and now: an introduction and historical reassessment. *N.T.M.*, v. 27, p. 407–417, 2019.

BUCELLI, M. et al. *A mathematical model that integrates cardiac electrophysiology, mechanics and fluid dynamics: application to the human left heart*. 2022.

BURCH, C. Logisim: A graphical system for logic circuit design and simulation. *ACM Journal of Educational Resources in Computing*, v. 2, p. 5–16, 03 2002.

CUELLAR, A. A. et al. Cellml 1.1 for the definition and exchange of biological models. *IFAC Proceedings Volumes*, v. 36, n. 15, p. 451–456, 2003. ISSN 1474-6670. 5th IFAC Symposium on Modelling and Control in Biomedical Systems 2003, Melbourne, Australia, 21-23 August 2003. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1474667017335450>.

ELLIS, T. O.; HEAFNER, J. F.; SIBLEY, W. L. *The GRAIL Language and Operations*. Santa Monica, CA: RAND Corporation, 1969.

FORTMANN-ROE, S. Insight maker: A general-purpose tool for web-based modeling and simulation. *Simulation Modelling Practice and Theory*, v. 47, p. 28–45, 09 2014.

HEINER, M. et al. Snoopy – a unifying petri net tool. In: HADDAD, S.; POMELLO, L. (Ed.). *Application and Theory of Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 398–407. ISBN 978-3-642-31131-4.

HEINER, M.; RICHTER, R.; SCHWARICK, M. Snoopy: A tool to design and animate/simulate graph-based formalisms. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008. (Simutools '08), p. 15:1–15:10. ISBN 978-963-9799-20-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1416222.1416242>.

HLADISH, T. et al. Epifire: An open source c++ library and application for contact network epidemiology. *BMC bioinformatics*, v. 13, p. 76, 05 2012.

LIU, F. *Colored Petri Nets for systems biology*. Tese (doctoralthesis) — BTU Cottbus - Senftenberg, 2012.

MALONEY, J. et al. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, Association for Computing Machinery, New York, NY, USA, v. 10, n. 4, nov 2010. Disponível em: <https://doi.org/10.1145/1868358.1868363>.

MARTINS, M.; FERREIRA, S.; VILELA, M. Multiscale models for biological systems. *Current Opinion in Colloid and Interface Science*, v. 15, n. 1, p. 18–23, 2010. ISSN 1359-0294. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1359029409000314>.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, v. 2014, n. 239, p. 2, 2014.

MORARU, I. et al. Virtual cell modelling and simulation software environment. *Systems Biology, IET*, v. 2, p. 352 – 362, 10 2008.

REIS, R. F. et al. A validated mathematical model of the cytokine release syndrome in severe covid-19. *Frontiers in Molecular Biosciences*, Frontiers, p. 680, 2021.

SCHAFF, J. et al. A general computational framework for modeling cellular structure and function. *Biophysical Journal*, v. 73, n. 3, p. 1135–1146, 1997. ISSN 0006-3495. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0006349597781463>.

SHANNON, P. et al. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, Cold Spring Harbor Lab, v. 13, n. 11, p. 2498–2504, 2003.

SPENCER, S. L. et al. An ordinary differential equation model for the multistep transformation to cancer. *Journal of Theoretical Biology*, Elsevier, v. 231, n. 4, p. 515–524, 2004.

TAKALA, T.; MäKäRäINEN, M.; HAMALAINEN, P. Immersive 3d modeling with blender and off-the-shelf hardware. In: . [S.l.: s.n.], 2013. p. 191–192. ISBN 978-1-4673-6097-5.

TALKINGTON, A.; DANTOIN, C.; DURRETT, R. Ordinary differential equation models for adoptive immunotherapy. *Bulletin of mathematical biology*, Springer, v. 80, p. 1059–1083, 2018.

VIGMOND, E. et al. Solvers for the cardiac bidomain equations. *Progress in Biophysics and Molecular Biology*, v. 96, n. 1, p. 3–18, 2008. ISSN 0079-6107. Cardiovascular Physiome. Disponível em: <https://www.sciencedirect.com/science/article/pii-/S0079610707000740>.

VIRTANEN, P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, v. 17, p. 261–272, 2020.

WALPOLE, J.; PAPIN, J. A.; PEIRCE, S. M. Multiscale computational models of complex biological systems. *Annual review of biomedical engineering*, v. 15, p. 137–54, 2013. Disponível em: <https://api.semanticscholar.org/CorpusID:39319868>.

WILENSKY, U. *NetLogo Software*. 2015. Disponível em: <http://ccl.northwestern.edu-/netlogo/>.

Apêndices

.1 Código de Python gerado para o modelo Predador-Presa

Código completo

```python
import argparse, contextlib, sys, os
import scipy
import numpy as np

def initial_values() -> np.ndarray:
    H_0 = 100.0
    P_0 = 5.0
    return np.array((
        H_0,
        P_0,
        ))


def constants() -> list:
    a = 0.015
    b = 0.0125
    m = 0.8
    r = 0.2
    return [
        a,
        b,
        m,
        r,
        ]


def variable_names() -> list[str]:
    return [
        "H",
        "P",
        ]


def system(t: np.float64, y: np.ndarray, *constants) -> np.ndarray:
    # populations
    H,P, = y

    # constants
    a,b,m,r, = constants

    dH_dt = (r * H ) - (a * (H * P ) )
    dP_dt = ((H * P ) * b ) - (P * m )

    return np.array([dH_dt,dP_dt])
```

```
45
46
47 def simulation_output_to_csv(sim_steps, simulation_output, write_to) ->
       str:
48     if not simulation_output.success:
49         print(simulation_output.message)
50         return
51
52     populatio_values_per_dt = simulation_output.y.T
53
54     write_to.write(f"t,{','.join(variable_names())}\n")
55
56     for dt, y in zip(sim_steps, populatio_values_per_dt):
57         write_to.write(f"{dt},")
58         write_to.write(",".join(f"{val:.4f}" for val in y))
59         write_to.write("\n")
60
61
62 def plot_simulation(sim_steps, simulation_output, filename):
63     import matplotlib.pyplot as plt
64     from matplotlib.backends.backend_pdf import PdfPages
65
66     with PdfPages(filename) as pdf:
67         # All
68         all_fig, all_ax = plt.subplots()
69         all_ax.grid()
70         all_ax.set(title=__name__, xlabel="Time", ylabel="Concentration"
       )
71
72         # Individually
73         for variable_name, variable_line_data in zip(variable_names(),
       simulation_output.y):
74             fig, ax = plt.subplots()
75             ax.grid()
76
77             ax.set(
78                 title=variable_name,
79                 xlabel="Time",
80                 ylabel="Concentration",
81             )
82
83             ax.plot(simulation_output.t, variable_line_data)
84             all_ax.plot(simulation_output.t, variable_line_data)
85
86             pdf.savefig(fig)
87
88         pdf.savefig(all_fig)
```

```
89
90
91 def file_or_stdout(filename: str | None):
92     if filename:
93         return open(filename, 'w')
94     else:
95         return sys.stdout
96
97
98 def simulate(filename, st=0, tf=10, dt=0.1, plot=False):
99     sim_steps = np.arange(st, tf + dt, dt)
100
101     simulation_output = scipy.integrate.solve_ivp(
102         fun=system,
103         t_span=(0, tf + dt),
104         y0=initial_values(),
105         args=constants(),
106         t_eval=sim_steps,
107     )
108
109     if plot:
110         plot_simulation(sim_steps, simulation_output, filename)
111
112     else:
113         with file_or_stdout(filename) as f:
114             simulation_output_to_csv(sim_steps, simulation_output, f)
115
116
117 if __name__ == "__main__":
118     parser = argparse.ArgumentParser()
119
120     parser.add_argument("--st", type=float, default=0)
121     parser.add_argument("--tf", type=float, default=10)
122     parser.add_argument("--dt", type=float, default=0.1)
123     parser.add_argument("-o", "--output", default=None)
124     parser.add_argument("--csv", action=argparse.BooleanOptionalAction)
125
126     args = parser.parse_args()
127
128     if args.output is None and not args.csv:
129         parser.error("when plotting (a.k.a --no-csv), an output file
    name is required via --output")
130
131     if args.output:
132         dirs = os.path.dirname(args.output)
133
134         if dirs:
```

```
135             os.makedirs(dirs, exist_ok=True)
136
137     simulate(args.output, plot=not args.csv)
```

.2 RI do modelo Predador-Presa

Arquivo JSON


```
{
  "metadata": {
    "name": "TODO",
    "type": "ode",
    "start_time": 0.0,
    "delta_time": 0.0,
    "end_time": 0.0,
    "positions": {
      "Eq/Predador": {
        "x": 962.0,
        "y": 702.0
      },
      "Eq/Presa": {
        "x": 984.0,
        "y": -10.0
      },
      "H": {
        "x": 159.0,
        "y": 115.0
      },
      "Mortalidade": {
        "x": 576.1239013671875,
        "y": 714.0438232421875
      },
      "P": {
        "x": 165.0,
        "y": 328.0
      },
      "Predacao": {
        "x": 497.0,
        "y": 329.0
      },
```

```
      "Predacao/Predador": {
        "x": 743.1011962890625,
        "y": 462.2484130859375
      },
      "Predacao/Presa": {
        "x": 750.0,
        "y": 239.0
      },
      "Reproducao": {
        "x": 547.0,
        "y": -40.0
      },
      "a": {
        "x": 518.0,
        "y": 198.0
      },
      "b": {
        "x": 518.0,
        "y": 570.0
      },
      "dH/dt": {
        "x": 1161.0,
        "y": 212.0
      },
      "dP/dt": {
        "x": 1160.0,
        "y": 374.0
      },
      "m": {
        "x": 167.0,
        "y": 697.0
      },
      "r": {
        "x": 289.0,
        "y": -70.0
      }
    }
  },
  "arguments": [
```

```
    {
      "name": "Predacao/Predador",
      "operation": "*",
      "composition": [
        {
          "name": "Predacao",
          "contribution": "+"
        },
        {
          "name": "b",
          "contribution": "+"
        }
      ]
    },
    {
      "name": "Predacao",
      "operation": "*",
      "composition": [
        {
          "name": "H",
          "contribution": "+"
        },
        {
          "name": "P",
          "contribution": "+"
        }
      ]
    },
    {
      "name": "Eq/Predador",
      "operation": "-",
      "composition": [
        {
          "name": "Predacao/Predador",
          "contribution": "+"
        },
        {
          "name": "Mortalidade",
          "contribution": "+"
```

```
111           }
112         ]
113       },
114       {
115         "name": "r",
116         "value": 0.2
117       },
118       {
119         "name": "P",
120         "value": 5.0
121       },
122       {
123         "name": "b",
124         "value": 0.0125
125       },
126       {
127         "name": "Mortalidade",
128         "operation": "*",
129         "composition": [
130           {
131             "name": "P",
132             "contribution": "+"
133           },
134           {
135             "name": "m",
136             "contribution": "+"
137           }
138         ]
139       },
140       {
141         "name": "Reproducao",
142         "operation": "*",
143         "composition": [
144           {
145             "name": "r",
146             "contribution": "+"
147           },
148           {
149             "name": "H",
```

```
150             " contribution ": "+"
151           }
152         ]
153       },
154       {
155         "name": "m",
156         "value": 0.8
157       },
158       {
159         "name": "a",
160         "value": 0.015
161       },
162       {
163         "name": "Predacao/Presa",
164         " operation ": "*",
165         " composition ": [
166           {
167             "name": "a",
168             " contribution ": "+"
169           },
170           {
171             "name": "Predacao",
172             " contribution ": "+"
173           }
174         ]
175       },
176       {
177         "name": "Eq/Presa",
178         " operation ": "-",
179         " composition ": [
180           {
181             "name": "Reproducao",
182             " contribution ": "+"
183           },
184           {
185             "name": "Predacao/Presa",
186             " contribution ": "+"
187           }
188         ]
```

```
    },
    {
      "name": "H",
      "value": 100.0
    }
  ],
  "equations": [
    {
      "name": "dP/dt",
      "operates_on": "P",
      "argument": "Eq/Predador",
      " contribution ": "+"
    },
    {
      "name": "dH/dt",
      "operates_on": "H",
      "argument": "Eq/Presa",
      " contribution ": "+"
    }
  ]
}
```