UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE EDUCAÇÃO SUPERIOR DO ALTO VALE DO ITAJAÍ – CEAVI
ENGENHARIA DE SOFTWARE

JÉSSICA BERNARDI PETERSEN

PORTABILIDADE DO MODELO DE PROPAGAÇÃO DE DOENÇAS A PARTIR DA ABORDAGEM DE DESENVOLVIMENTO DIRIGIDO A MODELOS COM AGENTES (MDD4ABMS)

IBIRAMA

2021

JÉSSICA BERNARDI PETERSEN

PORTABILIDADE DO MODELO DE PROPAGAÇÃO DE DOENÇAS A PARTIR DA ABORDAGEM DE DESENVOLVIMENTO DIRIGIDO A MODELOS COM AGENTES (MDD4ABMS)

Trabalho de conclusão apresentado ao curso de Engenharia de Software do Centro de Educação Superior do Alto Vale do Itajaí (CEAVI), da Universidade do Estado de Santa Catarina (UDESC), como requisito parcial para a obtenção do grau de bacharel em Engenharia de Software.

Orientador: Prof. Dr. Fernando dos Santos

IBIRAMA

2021

JÉSSICA BERNARDI PETERSEN

PORTABILIDADE DO MODELO DE PROPAGAÇÃO DE DOENÇAS A PARTIR DA ABORDAGEM DE DESENVOLVIMENTO DIRIGIDO A MODELOS COM AGENTES (MDD4ABMS)

Trabalho de conclusão apresentado ao curso de Engenharia de Software do Centro de Educação Superior do Alto Vale do Itajaí (CEAVI), da Universidade do Estado de Santa Catarina (UDESC), como requisito parcial para a obtenção do grau de bacharel em Engenharia de Software.

Banca Examinadora

Orientador:

Prof. Dr. Fernando dos Santos
UDESC

Membros:

Prof. Dra. Marilia Guterres Ferreira
UDESC

Prof. Dr. Adilson Vahldick
UDESC

Ibirama, 07/04/2021

Dedico esse trabalho aos meus pais, Márcia e Carlos que sempre me apoiaram em tudo, aos meus Irmãos, Conrado e Gabriel, aos meus professores e à UDESC Alto Vale.

AGRADECIMENTOS

Agradeço aos meus pais maravilhosos, Carlos e Márcia que sempre estão ao meu lado, me apoiando e incentivando, sem vocês não teria chegado tão longe. Eu Amo Vocês.

Agradeço aos meus irmãos e minhas cunhadas, por todo amor e carinho recebido. Eu Amo Vocês.

Agradeço às minhas cachorrinhas Mel e Serena por sempre me receberem em casa com alegria e muitos beijos, também conhecidos como lambidas. Eu Amo Vocês.

Agradeço aos meus amigos que considero como irmãos, Alexander, Aparício e Lucas, que entraram comigo na faculdade e que estiveram ao meu lado ao longo destes anos, rindo, se divertindo, jogando cs e comendo pizza.

Agradeço a todos os professores e funcionários da UDESC por auxiliar na minha formação academica. Entre eles agradeço ao professor Paulo Roberto Farah por ter sido meu orientador na monitoria de Introdução a programação, ao professor Marcelo de Souza por ter sido meu orientador de estágio na IPM Sistemas, ao professor Paolo Moser por ter sido meu orientador de monitoria das disciplinas de matemática, ao professor Adilson Vahldick por ter sido meu orientador de estágio obrigatório e principalmente ao Professor Dr. Fernando dos Santos por ter sido meu orientador de monitoria de Sistemas Distribuídos e Paralelos e orientador do presente TCC.

Agradeço a toda minha familia e todas as pessoas que estão e/ou já estiveram presentes na minha vida e já contribuiram com meu aprendizado.

Agradeço aos meus guias espirituais, também chamados de anjos da guarda, por sempre tentarem me mostrar o melhor caminho.

Aqui vai uma mensagem que achei na internet para todos vocês citados:
"Dizer obrigada, às vezes, não é suficiente para agradecer a tão amável e gentil pessoa que nos momentos das nossas vidas, aqueles mais difíceis, nos estende a mão amiga e nos oferece amparo. Estou agradecida a você e não sei neste instante como retribuir tanto carinho, mas é claro que encontrarei uma maneira de fazê-lo. Estou à sua disposição para quando precisar, a qualquer momento e a qualquer hora. Estendo-lhe minhas mãos, segure-as se precisar. E, obrigada por tudo!"

RESUMO

O presente trabalho apresenta a portabilidade da habilidade de propagação de doenças de simulações com agentes desenvolvidas através da Abordagem Dirigida a Modelos para Simulações com Agentes (MDD4ABMS). A MDD4ABMS disponibiliza uma ferramenta denominada ABStractme que permite ao projetista especificar a simulação com agentes através de elementos disponíveis na paleta de componentes além de possuir um gerador automático de código fonte para a plataforma NetLogo e recentemente para a plataforma Repast, porém essa última ainda não está completa. Portanto, com a portabilidade desenvolvida neste trabalho foi possível gerar código fonte de forma automática da habilidade de propagação de doenças para a plataforma de simulação Repast, permitindo aos desenvolvedores que possuem conhecimento nessa plataforma possam gerar código e executar as simulações de propagação de doença especificadas através da abordagem MDD4ABMS. Desta forma, é possível gerar código fonte, a partir de um mesmo modelo projetado na ferramenta ABStractme, para as plataformas Repast e NetLogo, permitindo realizar comparações entre elas. Um estudo de caso foi realizado considerando dois tipos de modelos de propagação de doenças com dois tipos de agente, que foram projetadas na ferramenta ABStractme e tiveram o código fonte Repast e NetLogo gerados de forma automática por meio da portabilidade desenvolvida nesse trabalho. Os resultados obtidos com a execução das simulações dos modelos em ambas plataformas foram similares, evidenciando a viabilidade da portabilidade de simulações com agentes desenvolvidas com a abordagem MDD4ABMS.

Palavras-chave: Simulações com agentes. MDD4ABMS. Portabilidade. Repast. Gerador de códigos

ABSTRACT

This work presents the portability of the ability of disease spread of agent-based simulations specified with the Model-driven Development of Agent-Based Simulations approach (MDD4ABMS). MDD4ABMS provides a tool called ABStractme that allows the designer specified the agent simulations through elements available in the component palette beside to have a code generator for the Netlogo plataform and recently to the Repast simulation too, but this last is not yet complete. Thus, with the portability developed in this work allowed to automatically generate source code of the ability of disease spread for Repast plataforms, allowing developers who have knowledge on that platform generate and execute the spread disease simulations specified through the MDD4ABMS approach. Therefore, it's possible to generate source code, from the same model designed in the ABStractme tool, to the Repast and Netlogo plataforms. A case study was performed considering the two models of disease spread with two agents, that were designed in the ABStractme tool and had the Repast and NetLogo source code generated automatically by means of the portability developed in this work. Results obtained with the execution of theses simulations in both plataforms were similar, giving evidence regarding the viability of porting simulations of the disease spread developed with the MDD4ABMS approach.

Keywords: Agent-based simulations. MDD4ABMS. Portability. Repast.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABMS	Agent-Based Modeling and Simulation
ABS	Agent-Based Simulation
GIS	Geographic Information Systems
HTML	Hypertext Markup Language
MDD	Model-Driven Development
MDD4ABMS	Model Driven Development for Agent-based Modeling and Simulation
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Uma simulação baseada em agente (*ABS - Agent-Based Simulation*) utiliza agentes simulados para reproduzir um fenômeno a ser analisado (KLÜGL; BAZZAN, 2012). Nesse paradigma, os agentes são entidades autônomas podendo representar pessoas e animais, que possuem atributos próprios como tempo de vida, estado de saúde e capacidade de multiplicação, que se comportam de maneira complexa e interagem entre si e com o ambiente ao longo do tempo na busca de atender seus objetivos.

Os agentes devem se adaptar às transformações ocorridas no ambiente e nos demais agentes ao decorrer da simulação. Para isso é necessário que seus estados sucessivos sejam planejados para conseguir decidir por si próprio o seu destino (BARROS et al., 2011). Desta forma, é possível analisar, por exemplo, a evacuação dos agentes em um ambiente de incêndio e a propagação de uma epidemia ou até de uma pandemia.

No início do ano de 2020, a doença conhecida como COVID-19, que se iniciou na China, chegou no mundo todo, virando uma pandemia. O Reino Unido optou por fazer o chamado *lockdown* devido ao resultado mostrado no modelo epidemiológico baseado em agentes desenvolvido por Neil Ferguson e sua equipe no Imperial College de Londres. A simulação modela a propagação da doença no nível dos indivíduos e de seus contatos, e previu, a menos que medidas fossem tomadas, que a doença causaria mais de meio milhão de mortes no Reino Unido (WOOLDRIDGE, 2020).

No paradigma de Modelagem e Simulação Baseadas em Agentes (*ABMS - Agent-Based Modeling and Simulation*), conforme Klügl e Bazzan (2012), as entidades são modeladas como agentes que interagem entre si e com o ambiente para reproduzir ou explorar algum fenômeno de estudo. Por conseguir aderir à complexidade decorrente do comportamento individual e interações existentes no mundo real, o ABMS tem sido usado em muitas áreas de simulações como tráfego, ecologia e epidemiologia (MACAL; NORTH, 2014).

Simulações com agentes geralmente são desenvolvidas em plataformas específicas para simulações que possuem bibliotecas para implementar o código fonte dos agentes e executar a simulação. Cada plataforma possui sua linguagem de programação, sendo então necessário uma pessoa que saiba programar naquela linguagem. Assim o Desenvolvimento Dirigido a Modelos (MDD) simplificou o processo de desenvolvimento desse tipo de simulação (SELIC, 2003).

O MDD utiliza modelos gráficos e componentes pré-criados para a construção visual de aplicativos complexos (MENDIX, 2020). A sua ideia principal, segundo Buarque (2012)

é transformar modelos de maiores níveis de abstração em modelos mais concretos até obter o código final, proporcionando ao projetista uma maior produtividade, portabilidade, menor custo pois irá diminuir o tempo gasto na programação além de permitir realizar modificações através do modelo abstrato.

Na abordagem MDD, o modelo é uma abstração criada com os elementos de modelagem relacionados à área de aplicação de simulação com agentes, onde os metamodelos devem estar relacionados com o agente, que ao final deve gerar o código fonte correspondente ao modelo abstrato (SELIC, 2003).

Assim, surgiu uma abordagem MDD para ABMS chamada Desenvolvimento Dirigido a Modelos para Simulações Baseadas em Agentes (MDD4ABMS) que possui como foco a definição de um metamodelo abstrato que captura os aspectos que compõe a ABS além da especificação da simulação (SANTOS, 2019).

O código pronto para execução é gerado automaticamente a partir de um transformador de modelo em código, independente da plataforma de simulação (SANTOS, 2019). A modelagem dos modelos de simulações é feita em um nível mais alto de abstração sem a necessidade de programação (SANTOS; NUNES; BAZZAN, 2017).

A abordagem MDD4ABMS original de Santos (2019) gera códigos fontes a partir de seus metamodelos somente para a plataforma de simulação NetLogo, dificultando seu uso por parte de projetistas que utilizam outras plataformas, como o Repast. O Repast é um dos *toolkits* gratuitos disponíveis para simulação baseada em agentes (NORTH; COLLIER; VOS, 2006).

Com base nessas limitações Tenfen (2019) propôs no seu trabalho desenvolver a portabilidade de modelos especificados através da abordagem MDD4AMBS para a plataforma de simulação Repast e sugeriu como trabalhos futuros a portabilidade do modelo de propagação de doenças. O modelo de propagação de doenças define como elas podem se alastrar em uma população. Na MDD4ABMS, o modelo de propagação de doenças é a capacidade em que o agente possui de transmitir uma doença a outro agente. Baseado nessa ideia, a proposta desse trabalho é justamente desenvolver essa portabilidade utilizando a abordagem MDD4ABMS.

1.1 PROBLEMA

Simulações modeladas com a abordagem MDD4ABMS, atualmente, geram executável para ser executado na plataforma NetLogo e recentemente foi desenvolvido a portabilidade para a plataforma Repast. Porém a portabilidade do modelo de propagação de doenças para essa plataforma ainda não foi desenvolvida. Portanto, o problema conside-

rado é a portabilidade do modelo de propagação de doenças para a plataforma Repast. Por ser a habilidade mais importante na área de epidemiologia, é imprescindível a portabilidade desse modelo para que os usuários da plataforma Repast possam utilizá-lo nas simulações que forem especificadas através da MDD4ABMS.

1.2 OBJETIVOS

Nesta seção se encontra o Objetivo Geral e os Objetivos Específicos.

1.2.1 Objetivo Geral

Oferecer portabilidade dos modelos de simulação de propagação de doenças da MDD4ABMS a partir do desenvolvimento de um gerador de código para a plataforma de simulação com agentes Repast.

1.2.2 Objetivos Específicos

a. Identificar e mapear os elementos do modelo de propagação de doenças para desenvolver simulações na plataforma Repast;

b. Desenvolver um gerador de código para gerar código da plataforma Repast;

c. Integrar o gerador de código com a abordagem MDD4ABMS.

1.3 JUSTIFICATIVA

Existem diversas ferramentas de simulações baseadas em agentes, como Repast, Swarm, NetLogo e StarLogo, em que todas partem do mesmo princípio: agentes que irão se relacionar com o ambiente e entre si, além de compartilhar recursos. Tendo em vista que os agentes compartilham o mesmo princípio e que cada uma das plataformas existentes adotam uma linguagem de programação própria, é possível gerar uma solução baseada em modelos abstratos através da compactação e da redução dos princípios do desenvolvimento de simulações baseadas em agentes. Modelos abstratos são modelos com alto grau de generalização.

O modelo de propagação de doenças é uma das habilidades dos agentes (SANTOS, 2019). Essa habilidade é muito utilizada na área epidemiológica. Partindo dos modelos abstratos, é possível gerar código fonte para diferentes plataformas a partir de um único

metamodelo, permitindo maior reusabilidade dos modelos de simulações entre as plataformas de simulações, além de aumentar a produtividade e reduzir os custos de produção.

1.4 HIPÓTESE

Com a abordagem MDD4ABMS será possível obter portabilidade do modelo de propagação de doenças a partir do desenvolvimento de um gerador de código fonte para a plataforma de simulação com agentes Repast.

A hipótese é que o gerador de código do presente trabalho gerará um executável do modelo de propagação de doenças pronto para rodar na plataforma Repast, permitindo ao projetista executar a simulação na plataforma em que possuir maior conhecimento.

1.5 METODOLOGIA

Forão realizadas as seguintes etapas para desenvolver o gerador de código fonte:

1. Identificação dos elementos do modelo de propagação de doenças disponíveis para modelagem na abordagem MDD4ABMS;
2. Criação das funcionalidades correspondente da abordagem MDD4ABMS com a biblioteca de objetos Java do Repast;
3. Compor as dependências necessárias da Repast à abordagem MDD4ABMS;
4. Mapear as funcionalidades da abordagem MDD4ABMS e da Repast para gerar código fonte utilizando as linguagens Xpand.
5. Realiza validações dos códigos gerados com Xpand.

Após as etapas acima, foi realizado um estudo sobre a abordagem MDD4ABMS de Santos (2019) e o *plugin* de modelagem ABStractme disponibilizada pela abordagem MDD4ABMS.

Também foi necessário aprender as funcionalidades e objetos disponíveis da biblioteca Java, fornecida pela plataforma Repast para realizar o desenvolvimento das simulações com agentes e a visualização das interações entre os agentes e o ambiente. Para assim, ser possível mapear os elementos da abordagem MDD4ABMS com as funcionalidades disponíveis no Repast.

As validações foram realizadas para garantir que todos os blocos estejam desempenhando a finalidade ao qual foram implementados e se estão integradas adequadamente.

Essas validações foram feitas através de depurações e inspeções detalhadas do código gerado pelas regras de produção de código.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção será abordada a simulação de agentes, bem como a abordagem MDD4ABMS além das tecnologias e ferramentas utilizadas no desenvolvimento deste trabalho.

2.1 SIMULAÇÕES COM AGENTES

Um agente, segundo Russel e Norvig (2004), é uma entidade que é capaz de perceber seu ambiente através de sensores, sendo capaz de agir sobre esse ambiente por meio de atuadores. Ao perceber o estado do seu ambiente, um agente deve tomar uma decisão para alcançar seu objetivo, para isso é necessário que seu funcionamento interno tenha autonomia, possua os objetivos a alcançar e tenha sua implementação, a qual pode adotar técnicas de inteligência artificial para aprimorar seu comportamento e decisões (SOUZA, 2017).

Além de interagir com o ambiente, os agentes podem interagir entre si, trocando dados, negociando, cooperando, entre outras atividades (WOOLDRIDGE, 2009). Conforme Klügl e Bazzan (2012), a ideia central do paradigma de simulação com agentes é usar agentes simulados para entender, predizer e reproduzir algum fenômeno de estudo. Esse paradigma possui alguns benefícios em relações aos outros, que são: (1) possibilidade de estudar diferentes estratégias de tomadas de decisões além de poder considerar a heterogeneidade dos agentes e (2) permitir estudar os resultados que são gerados a partir das ações e interações entre os agentes e com o ambiente (KLÜGL, 2008).

Na criação de uma simulação com agentes é necessário que três elementos sejam tratados. O primeiro é o conjunto de agentes que devem ser autônomos em relação às outras entidades do ambiente. O segundo é a especificação da interação entre os agentes e com o ambiente que compartilham, visto que a interação deles é o que produzirá o resultado. Já o último elemento é o ambiente simulado que contém todos outros elementos (KLÜGL; BAZZAN, 2012).

Para exemplificar como funciona a interação entre os três elementos, Klügl e Bazzan (2012) citam um exemplo simples, em que os agentes são uma presa (ovelha) e um predador (lobo) e o elemento do ambiente é representado pela grama.

Ao interagirem, algum tipo de saída é gerado, que pode ser visualizado na Tabela 1, ou seja, um lobo ao interagir com um lobo, a ação de saída será a de reprodução. A mesma ação é gerada na iteração entre duas ovelhas. Já um lobo ao interagir com uma ovelha será gerado a ação de alimentar-se (lobo se alimenta da ovelha). Na interação ao

contrário, ovelha com lobo, a ovelha será devorada. O lobo não interage com a grama, já a ovelha vai se alimentar da grama.

Tabela 1 – Interação de um modelo presa-predador

	Lobo	*Ovelha*	*Grama*
Lobo	Reproduzir	Se alimentar	-
Ovelha	Ser devorado	Reproduzir	Se alimentar
Grama	-	-	-

Fonte: Klügl e Bazzan (2012)

2.2 ABORDAGEM MDD4ABMS

O Desenvolvimento Dirigido a Modelos (MDD) é uma abordagem para apoiar o desenvolvimento de software, em que são utilizados artefatos de modelagem de alto nível, como os diagramas UML, para impulsionar o desenvolvimento de artefatos de baixo nível, como o código-fonte (MERNIK; HEERING; SLOANE, 2005).

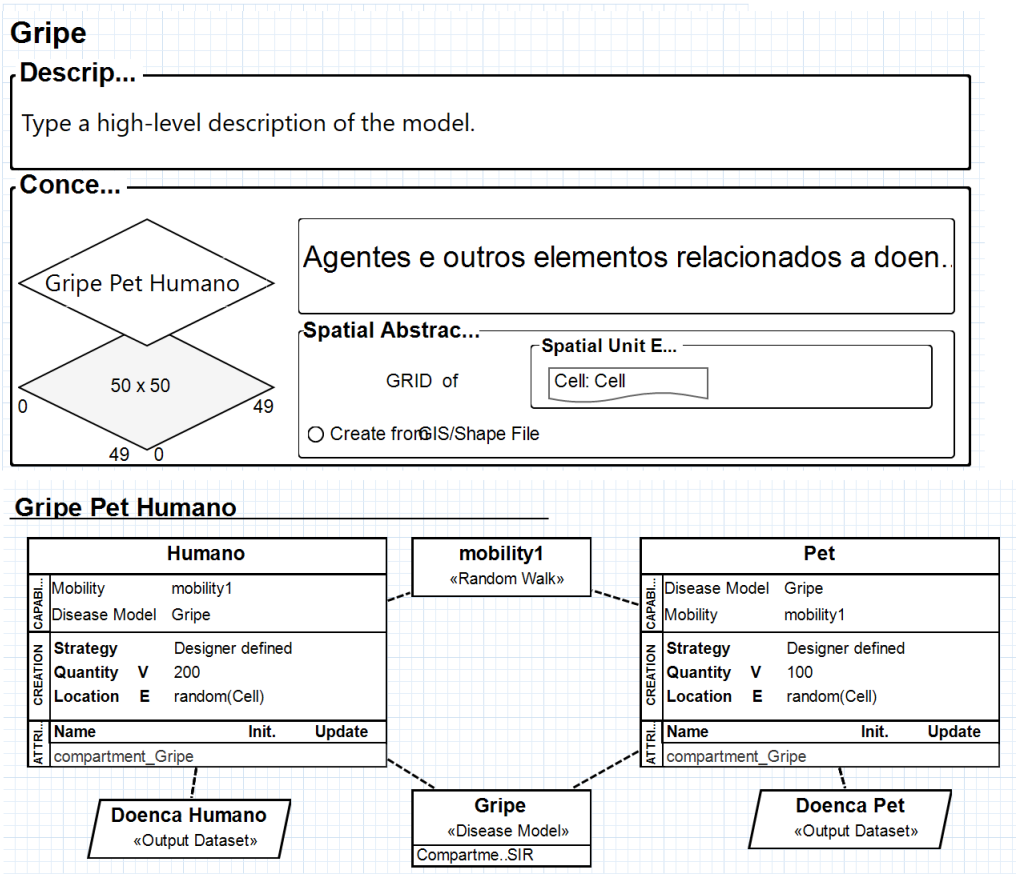
Uma abordagem MDD permite elevar um modelo a um nível mais abstrato, facilitando o entendimento de sistemas mais complexos (SELIC, 2003) onde devemos dizer quais funcionalidades devem ter ao invés de como fazer (ATKINSON; KÜHNE, 2003). A modelagem torna-se o papel chave no processo de desenvolvimento, pois é a partir do modelo que o código fonte será gerado automaticamente, tornando o processo mais produtivo e eficiente permitindo a reusabilidade e a portabilidade (MOHAGHEGHI; DEHLEN; NEPLE, 2009).

Uma abordagem que utiliza MDD para o desenvolvimento de simulações com agentes é a MDD4ABMS (SANTOS; NUNES; BAZZAN, 2018). Foram seguidas quatro atividades principais que compreendem o processo iterativo para a construção dessa abordagem: definição da sintaxe abstrata, definição da sintaxe concreta, definição da semântica dos elementos de linguagem e integração da linguagem com uma plataforma de execução (SANTOS; NUNES; BAZZAN, 2018).

A MDD4ABMS possui como foco a definição de um metamodelo abstrato que captura os aspectos que compõe a ABS bem como a especificação da simulação. A modelagem é feita em nível mais alto de abstração, não necessitando programar. O modelo da simulação é formado apenas pelas características do agente e detalhes da transmissão da doença. O código é gerado automaticamente a partir de um transformador de modelo em código (SANTOS; NUNES; BAZZAN, 2017).

A abordagem MDD4ABMS contém uma ferramenta de modelagem chamada ABStractme onde o projetista pode especificar a simulação com agentes através de elementos disponíveis na paleta de componentes. Seu ambiente de modelagem pode ser visto na Figura 1. A partir da modelagem é possível gerar códigos de modo automático para a execução da simulação (MOREIRA et al., 2017).

Figura 1 – Ambiente de modelagem da ferramenta ABStractme



Fonte: Elaborada pela autora.

A limitação dessa abordagem era a geração de código apenas para a plataforma de simulação NetLogo. Com o trabalho de Tenfen (2019) foi possível realizar a portabilidade de alguns aspectos da simulação para a plataforma Repast, como o ambiente de grid de duas dimensões com gráficos de saída, atribuição dos valores ao atributo do ambiente com base em arquivos GIS e o agente com as habilidades de mobilidade, sobrevivência e habilidade definida externamente além da atualização de atributos. A presente monografia propõe complementar o trabalho iniciado por Tenfen (2019) e desenvolver a portabilidade do modelo de propagação de doenças para a plataforma Repast.

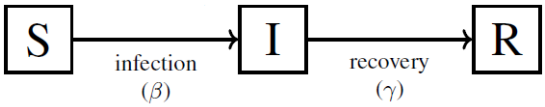
Epidemiologistas utilizam os modelos matemáticos de propagação de doenças para analisar como uma doença infecciosa afeta uma população particular de indivíduos e

quais as políticas podem ser adotadas para evitar ou controlar uma pandemia (ISERN; MORENO, 2016).

Simulações baseadas em agentes tem sido usadas para analisar a propagação de doenças. Esse tipo de simulação é influenciada pela heterogeneidade da população, limite de espaço, interação entre os indivíduos, por exemplo. Quando todos esses aspectos são levados em conta, políticas mais efetivas podem ser adotadas (SANTOS, 2019).

A MDD4ABMS permite especificar simulações de propagação de doenças. Segundo Santos (2019), é adotado o modelo compartimental de Kermack e McKendrick (1932). Esse modelo é muito utilizado por epidemiologistas para prever e entender a propagação de doenças. O modelo mais simples é chamado de SIR e classifica os indivíduos em compartimentos: S (suscetível) para indivíduos ainda não expostos à doença, I (Infectado) para indivíduos com a doença e R (Recuperado) para indivíduos que foram curados da doença. As transições entre os compartimentos representam a dinâmica da doença como mostrado na Figura 2. Essas transições são governadas pela taxa de transmissão (β) e taxa de recuperação (γ).

Figura 2 – Modelo epidemiológico compartimental SIR



Fonte: Santos (2019).

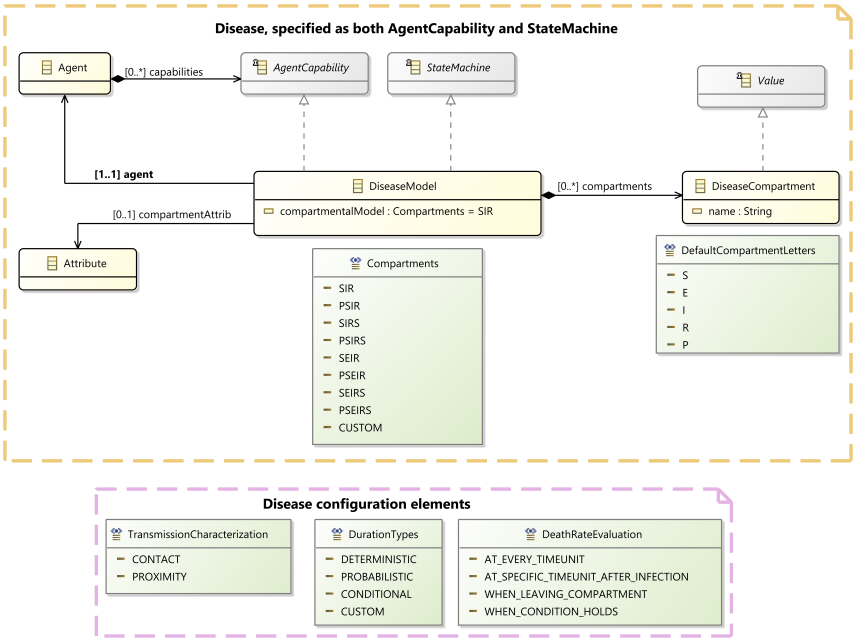
Outros parâmetros de doenças podem ser considerados, segundo Santos (2019), como as mortes causadas pela doença, que podem ser incorporadas por um taxa de mortalidade, e a imunidade temporária, que pode ser obtida pela especificação da duração do compartimento R, levando ao modelo chamado SIRS. Se considerar a imunidade passiva e o tempo de incubação da doença pode levar ao modelo PSEIR. A MDD4ABMS utiliza os modelos compartimentais criados como o SIR, SEIR e PSEIR ou pode-se criar um modelo compartimental através do modelo CUSTOM.

Nos modelos epidemiológicos contemplados pela abordagem MDD4AMBS são identificados dois aspectos na simulação de propagação de doenças: interações e inicialização. A interação especifica em que tipo de contato entre agente-agente ou entidade-agente a doença pode ser transmitida. Na inicialização são detalhados todos aspectos de como a doença começou a se propagar (SANTOS, 2019).

Na MDD4ABMS o modelo compartimental de doenças é abstraído como uma máquina de estados. Os estados são representados pelos compartimentos e as transições entre estados são as que estão entre os compartimentos. Na Figura 3 é mostrado o metamodelo

da máquina de estados *DiseaseModel*, que estende *StateMachine* e especifica os estados de acordo com os modelos compartimentais de propagação de doença (SANTOS, 2019).

Figura 3 – Metamodelo de doença como capacidade do agente



Fonte: MDD4ABMS (2017).

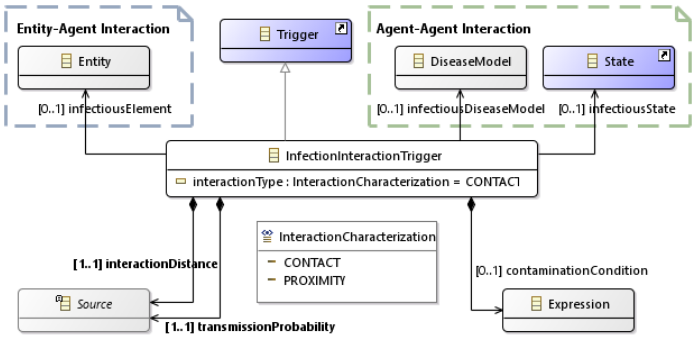
Dado que o agente, que está sujeito a uma doença, está no comando de seu modelo compartimental, a máquina de estado da doença também é um *AgentCapability*. Um *AgentCapability* abstrai a noção da capacidade do agente, como por exemplo mobilidade e sobrevivência. Os modelos compartimentais fornecidos são enumerados no elemento *Compartments*, sendo um deles atribuído à máquina de estado para especificar qual será o modelo compartimental adotado e por quais estados a máquina será composta (SANTOS, 2019).

Especializações de *State* são especificados, de acordo com Santos (2019), para representar os compartimentos e apenas esses estados especializados são aceitos no estado da máquina *DiseaseModel*. A máquina de estados de doença está associada com um atributo do agente que fornece acesso ao compartimento atual e é atualizado pelo estado da máquina.

A infecção causada pela interação social é especificada como uma transição do estado suscetível para o exposto ou infectado (Figura 4). Para infecções causadas por transmissões entre agentes, é utilizado o modelo de doenças do outro agente envolvido na interação e seu estado, assim, um agente pode se infectar sempre que estiver no estado suscetível e existir uma interação com outro agente no estado infeccioso, dependendo somente da

taxa de infecção. Já para infecções causadas pela interação com objetos contaminados, é referido como entidade infecciosa (SANTOS, 2019).

Figura 4 – Metamodelo de infecção



Fonte: Santos (2019).

No metamodelo da MDD4ABMS, uma interação, que é representada pelo elemento *InfectionInteractionTrigger*, é caracterizada pelo contato físico ou proximidade espacial. O atributo *interactionType* desse elemento especifica qual caracterização da interação é considerada pela transmissão da infecção. A probabilidade de transmissão é parte da interação e determina quando um estado da doença moverá para o estado de transição alvo ao ocorrer uma interação (SANTOS, 2019).

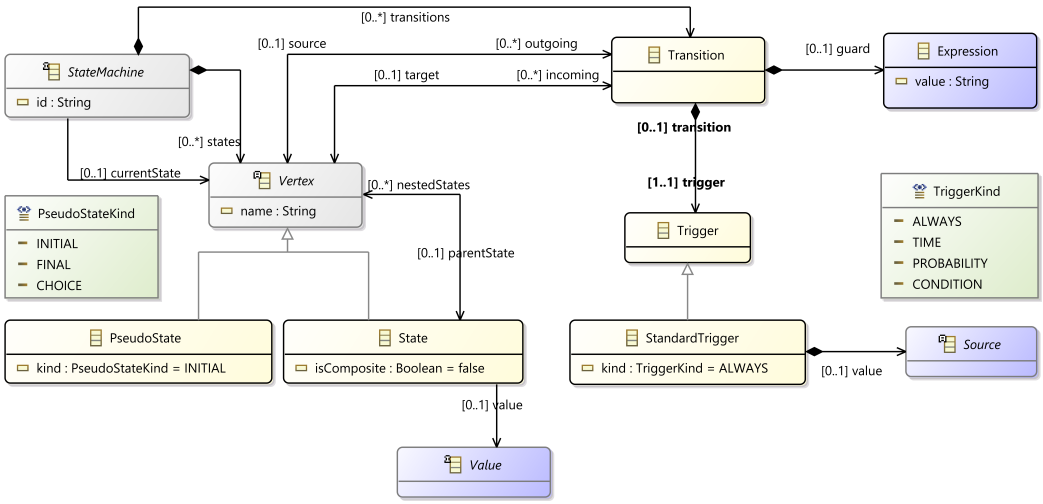
Após a infecção, o compartimento do agente atual é modificado de acordo com a progressão da doença, conforme a Figura 5. O modelo compartimental representa essa progressão como uma transição governada por taxas, sendo relacionadas com a duração de cada estado, representadas como compartimentos. A duração de um compartimento particular é especificada pelo tipo de *trigger* associado com sua *transition*, que também possui uma cláusula de *guard* (SANTOS, 2019).

O *kind* da *trigger* pode ser *always*, que será disparado sempre, *time* que é disparado após um período de tempo e possui um *value* da *trigger*, *probability* que é disparada probabilisticamente ou *condition* que é disparada quando a condição associada à *trigger* é satisfeita, todas requerem *guard* = *true* para transitar (SANTOS, 2019).

Para acrescentar outras semânticas, segundo Santos (2019), existe uma transição especializada chamada *ProgressionTransition*. Existem quatro modos distintos de especificar a duração de um compartimento: probabilístico, determinístico, condicional e duração customizada (CUSTOM).

Pelo modo probabilístico é especificada uma taxa de probabilidade, em que o agente só irá para o próximo compartimento ao atingir essa taxa. No determinístico, um período fixo de tempo é especificado e o agente permanece no compartimento por esse período. No condicional é especificado uma condição e o agente permanece até a condição ser satisfeita,

Figura 5 – Metamodelo de progressão e Mortalidade



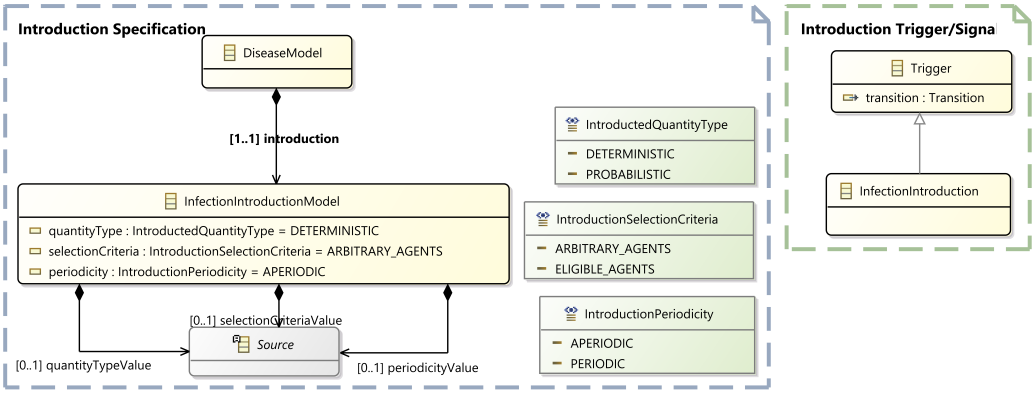
Fonte: MDD4ABMS (2017).

já a customizada é uma combinação dos anteriores. As mortes causadas pela doença são especificadas como *transition* para um pseudoestado adicional chamado *dead*, cujo seu tipo é *final*, onde o agente morre. Elas possuem uma *triggerKind* como *probability*, que corresponde a taxa de mortalidade (SANTOS, 2019).

O metamodelo da MDD4AMBS, de acordo com Santos (2019), prevê as quatro seguintes circunstâncias que levam a este compartimento. “a todo momento” onde a taxa de mortalidade é calculada a cada *tick* enquanto o agente permanecer no compartimento; “em um momento específico após infectado” onde a taxa de mortalidade é calculada em uma unidade de tempo específico, relativa ao momento em que o agente entrou no compartimento; “quando a condição ocorrer” , a taxa de mortalidade é calculada sempre que uma condição persistir e “ao sair do compartimento” a taxa de mortalidade é calculada apenas quando o tempo de permanência no compartimento é alcançada e o estado da máquina está se movendo para o próximo compartimento. As três primeiras são especificadas via condição *guard* enquanto que a última será via *pseudo-states* do tipo *choice*, sendo o único que não possui como o alvo da transição como *dead*. O tipo “quando a condição ocorrer” possui um *value* na *guard* para especificar o momento específico, caracterizado por *timeExpression*.

A introdução da doença está além do escopo do agente, pois, segundo Santos (2019), não é ele que decide se é ou não infectado. Esse aspecto é uma tarefa a ser executada pelo controle da simulação. Todos os aspectos que controlam a introdução da doença são especificadas como elementos *DiseaseIntroductionModel* que são acessados pelo controle da simulação (Figura 6).

Figura 6 – Metamodelo de introdução da doença



Fonte: MDD4ABMS (2017).

Dentro do elemento *DiseaseIntroductionModel* tem três atributos que indicam: a quantidade de agentes para serem infectados, o critério de seleção e a periodicidade. A quantidade poderá ser determinística, com uma quantidade exata de agentes ou probabilística, em que os agentes são selecionados de acordo com uma taxa de probabilidade. O critério de seleção pode ser aleatório ou elegível, em que os agentes serão selecionados de acordo com um critério. A periodicidade pode ser aperiódica, em que os agentes serão introduzidos uma única vez ou periódicas, em que serão introduzidos periodicamente (SANTOS, 2019).

Para permitir a troca de compartimento do agente selecionado para ser infectado, a máquina de estado de doença possui uma transição de compartimento suscetível para exposto ou infectado. A *trigger* especializada chamada *DiseaseIntroduction* é usada nessa transição, que captura um sinal de evento mandado pelo controle da simulação para a máquina de estado de doença quando um agente é selecionado e deve ser infectado (SANTOS, 2019).

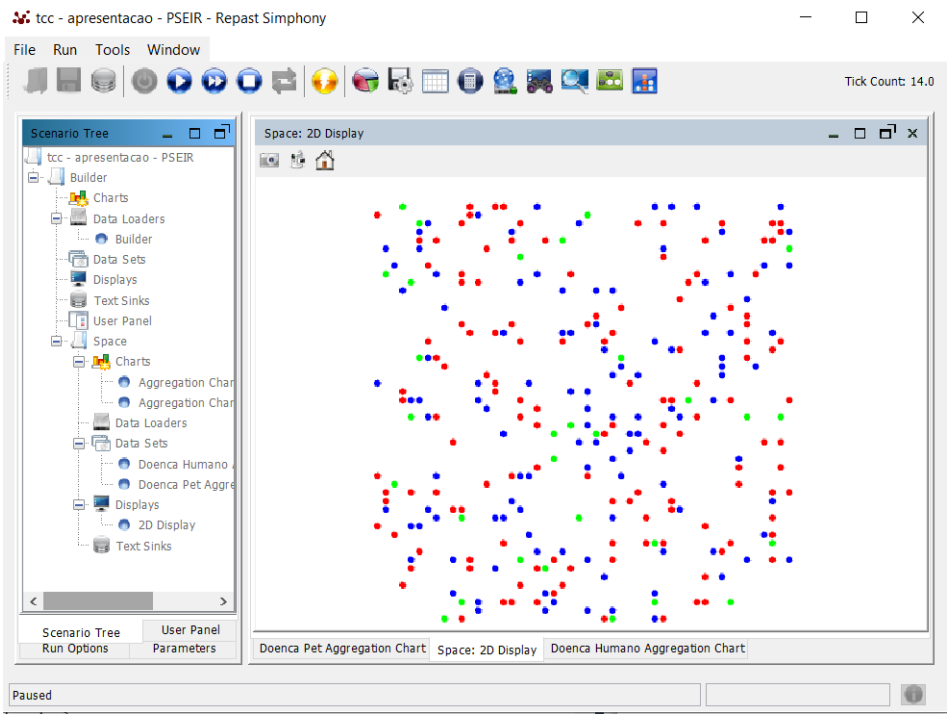
A abordagem MDD4ABMS, segundo Santos (2019), permite gerar gráficos de saída (*output*) que podem ser utilizados para analisar os resultados obtidos da simulação. Cada gráfico pode conter várias informações, no caso da doença, um gráfico pode representar os compartimentos da doença do agente. Para sua geração é necessário ter um critério de eligibilidade, no caso da doença, cada compartimento terá um critério de eligibilidade. Por exemplo, o agente Humano possui uma doença chamada Sarampo, que possui os compartimentos SIR, desta forma um dos critérios de eligibilidade será igual a *compartment_Sarampo = "S"*.

2.3 REPAST

O Repast foi originalmente desenvolvido por Collier, Sallach e outros (COLLIER; HOWE; NORTH, 2004) na Universidade de Chicago (REPAST, 2008). Esse *toolkit* de acordo com Lima et al. (2009) implementa a simulação como uma máquina de estados cujo estados são formado pelo conjunto de seus componentes. Esses componentes são divididos em: (1) infra-estrutura, que são os mecanismos que executam a simulação, exibem e coletam dados, e (2) representação, que é o próprio modelo de simulação.

É possível desenvolver modelos de Repast de diferentes formas, incluindo o ReLogo, Groovy e até Java, podendo ser intercalados com fluidez (REPAST, 2019). O ambiente de simulação do Repast pode ser visto na Figura 7. Segundo (COLLIER; NORTH, 2016), para desenvolver uma simulação com agentes pode-se apenas instanciar e utilizar os objetos da biblioteca do Repast, criando assim, o ambiente, a manipulação dos agentes, as interações e demais ações.

Figura 7 – Ambiente de simulação da plataforma Repast



Fonte: MDD4ABMS (2017).

Para explicar o Repast, vamos utilizar o exemplo de Collier e North (2016), no qual foi criada uma simulação com humanos e zumbis. O zumbi deve verificar qual localização ao seu redor possui maior número de humanos e se dirigir para aquela direção, que, ao tocar no humano, esse virará um zumbi. Já os humanos ao verificar que um zumbi se aproxima, deverá ir para o local com menor quantidade de zumbis.

O exemplo está implementado na linguagem Java. Para representar o ambiente em que os agentes humano e zumbi estão inseridos, suas classes possuem os atributos *ContinuousSpace* e *Grid*. Um *ContinuousSpace* nos permite utilizar pontos flutuantes e um *Grid* nos permite realizar consultas de vizinho e de proximidade usando coordenadas inteiras.

Na Figura 8 temos a classe *Zombie* que é responsável por todas ações realizadas pelos objetos *zombies*. Foi utilizado a anotação *@ScheduleMethod* no método *step()* para que a cada iteração de todos os objetos *zombies* esse método seja chamado.

Figura 8 – Classe Zombie em Java

```
1  public class Zombie {
2      private ContinuousSpace<Object> space;
3      private Grid<Object> grid;
4      private boolean moved;
5      public Zombie(ContinuousSpace<Object> space, Grid<Object< grid){
6          this.grid = grid;
7          this.space = space;
8      }
9      @ScheduleMethod(start = 1, interval = 1)
10     public void step(){
11         GridPoint pt = grid.getLocation(this);
12         GridCellNgh<Human> nghCreator = new GridCellNgh<Human>(grid, pt, Human.class, 1, 1);
13         List<GridCell<Human>> gridCells = nghCreator.getNeighborhood(true);
14         SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
15         GridPoint pointWithMostHumans = null;
16         int maxCount = -1;
17         for(GridCell<Human> cell: gridCells){
18             if(cell.size() > maxCount){
19                 pointWithMostHumans = cell.getPoint();
20                 maxCount = cell.size();
21             }
22         }
23         moveTowards(pointWithMostHumans);
24         infect();
25     }
26     private void moveTowards(GridPoint pointWithMostHumans){
27         if(!pointWithMostHumans.equals(grid.getLocation(this))){
28             NdPoint myPoint = space.getLocation(this);
29             NdPoint otherPoint = new NdPoint(pointWithMostHumans.getX(), pointWithMostHumans.getY());
30             space.moveByVector(this, 1, angle, 0);
31             myPoint = space.getLocation(this);
32             grid.moveTo(this, (int) myPoint.getX(), myPoint.getY());
33             moved = true;
34         }
35     }
36     public void infect(){
37         GridPoint pt = grid.getLocation(this);
38         List<Object> humans = new ArrayList<Object>();
39         for(Object obj: grid.getObjectsAt(pt.getX(), pt.getY())){
40             if(obj instanceof Human){
41                 humans.add(obj);
42             }
43         }
44         if(humans.size() > 0){
45             int index = RandomHelper.nextIntFromTo(0, humans.size(), -1);
46             Object obj = humans.get(index);
47             NdPoint spacePt = space.getLocation(obj);
48             context<Object> context = ContextUtils.getContext(obj);
49             context.remove(obj);
50             Zombie zombie = new Zombie(space, grid);
51             context.add(zombie);
52             space.moveTo(zombie, spacePt.getX(), spacePt.getY());
53             grid.moveTo(zombie, pt.getX(), pt.getY());
54             Network<Object> net = (Network<Object>) context.getProjection("infection network");
55             net.addEdge(this, zombie);
56         }
57     }
58 }
```

Fonte: Adaptado de Collier e North (2016).

No método *step* é armazenada a localização atual do próprio objeto *zombie*, em que é feita uma lista com as células vizinhas que possuem objetos do tipo *human*. A lista é embaralhada para que o *zombie* não ande sempre na mesma direção, caso as células sejam todas iguais, e então é verificada a célula com maior número de *human*, sendo passada para o método *moveTowards*.

No método *moveTowards*, é feita uma verificação para saber se o *zombie* já não está nessa célula. Caso não esteja, a localização do *zombie* é armazenada em um *NdPoint* (que armazena a coordenada como um *double*), a localização para onde o *zombie* deve se dirigir é convertida em *NdPoint*. Essa conversão é necessária para poder se mover em um *ContinuousSpace*. Em seguida, calcula-se o ângulo no qual o *zombie* terá que andar em uma unidade e a posição do *zombie* no *Grid* é atualizada.

Após a chamada do método *moveTowards*, o método *infect()* é chamado, no qual a posição da célula do *zombie* é armazenada em um *GridPoint* e é criada uma lista de objetos em que conterá todos os objetos da classe *Human* que estejam perto do *zombie* (se contiver algum por perto). Caso tenha objetos *humans* dentro da lista, um dos objetos da mesma é escolhido de forma aleatória para ser infectado. Ao ser infectado ele é removido da simulação e um novo *zombie* é adicionado a essa simulação. Uma linha entre o *zombie* que infectou e o novo *zombie* é então criada.

A classe *Human* (Figura 9) define um atributo inteiro de energia, que é gasta quando o agente se movimenta. Seu construtor é igual ao da classe *Zombie*, porém possui a adição da energia. No método de implementação da movimentação *run()*, terá a anotação *@Watch*, que será acionado sempre que um *zombie* entrar em sua vizinhança. O *Watch* irá observar por qualquer mudança na variável *"moved"* da classe *Zombie*, assim quando a variável mudar, o *Watcher* irá ser checado para todos objetos *humans*, assim, se retornar *true*, o método *run()* será acionado para aquele *humans*.

Sua implementação é igual ao do *Zombie*, possuindo apenas as diferenças de verificar qual é a célula que tem a menor quantidade de *zombies* (ao invés da maior quantidade), e também se ainda possui energia para então poder mover-se.

Não sendo possível realizar a movimentação, o *human* permanece no lugar e sua energia é reiniciada, caso tenha energia, ele conseguirá realizar a movimentação. Será então chamado o método *moveTowards* e passado como parâmetro a célula ao qual o *human* deve mover-se (com menor quantidade de *zombies*). O método *moveTowards* é igual ao do *Zombie* com exceção da energia que é decrementada a cada movimentação e ele anda duas células ao invés de apenas uma.

Para inicializar a simulação, é necessário que a classe de inicialização implemente a classe do Repast chamada *ContextBuilder*. Um *ContextBuilder* irá criar um *context*,

Figura 9 – Classe Human em Java

```
1  public class Human{
2      private ContinuousSpace<Object> space;
3      private Grid<Object> grid;
4      private int energy, startingEnergy;
5      public Human(ContinuousSpace<Object> space, Grid<Object> grid, int energy) {
6          this.space = space;
7          this.grid = grid;
8          this.energy = this.startingEnergy = energy;
9      }
10     @Watch(watcheeClassName = "jzombies.Zombie",
11                          watcheeFieldNames = "moved",
12                          query = "within_moore 1",
13                          whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
14     public void run(){
15         GridPoint pt = grid.getLocation(this);
16         GridCellNgh<Zombie> nghCreator = new GridCellNgh<Zombie>(grid, pt, Zombie.class,1,1);
17         List<GridCell<Zombie>> gridCells = nghCreator.getNeighborhood(true);
18         SimUtilities.shuffle(gridCells, RandomHelper.getUniform());
19         GridPoint pointWithLeastZombies = null;
20         int minCount = Integer.MAX_VALUE;
21         for(GridCell<Zombie> cell: gridCells){
22             if(cell.size() < minCount){
23                 pointWithLeastZombies = cell.getPoint();
24                 minCount = cell.size();
25             }
26         }
27         if(energy > 0){
28             moveTowards(pointWithLeastZombies);
29         }else{
30             energy = startingEnergy;
31         }
32     }
33     private void moveTowards(GridPointWithLeastZombies){
34         if(!pointWithLeastZombies.equals(grid.getLocation(this))){
35             NdPoint myPoint = space.getLocation(this);
36             NdPoint otherPoint = new NdPoint(GridPointWithLeastZombies.getX(),
37                                              GridPointWithLeastZombies.getY());
38             double angle = SpatialMath.calcAngleFor2DMovement(space, myPoint, otherPoint);
39             space.moveByVector(this, 2, angle, 0);
40             myPoint = space.getLocation(this);
41             grid.moveTo(this, (int) myPoint.getX(), (int) myPoint.getY());
42             energy--;
43         }
44     }
45 }
```

Fonte: Adaptado de Collier e North (2016).

que é composto por um conjunto de agentes (*zombies* e *humans*), podendo ter *Projections* associados. *ContinuousSpace* e *Grid* são exemplos de *Projections*. Será inicializado o *context*, criado os *Projections* e os agentes dentro do método *build*, como mostrado na Figura 10.

Esse método começa com a criação de um *network* para formar uma rede de nós entre os *zombies* e seus descendentes (*humans* infectados que vão virar *zombie*). Então o *context* é inicializado e as projeções são criadas. O id do *context* (linha 6) deve ser igual ao que será criado no arquivo *context.xml*. Esse arquivo *context* descreve a hierarquia do modelo, ou seja, a composição dos contextos que são usados no modelo e as projeções que

são associados a eles. No exemplo apresentado, temos um único *context* com o id *jzombies* e três projeções (*continuousSpace*, *grid* e *network*) que serão explicadas logo abaixo.

Após a criação do *ContinuousSpace* e do *Grid*, serão criados os agentes (*zombies* e *humans*) e estes adicionados dentro do *context*. Ao adicionar dentro do *context*, automaticamente são adicionados em todos os *projections* associados ao *context*.

Figura 10 – Classe inicializadora JZombiesBuilder em Java

```
1  public class JZombiesBuilder implements ContextBuilder<Object>{
2      @Override
3      public Context build(Context<Object> context){
4          NetworkBuilder<Object> netBuilder = new NetworkBuilder<Object>("infection network", context, true);
5          netBuilder.buildNetwork();
6          context.setId("jzombies");
7          ContinuousSpaceFactory spaceFactory = ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
8          ContinuousSpace<Object> space = spaceFactory.createContinuousSpace("space", context,
9                      new RandomCartesianAdder<Object>(),
10                     new repast.simphony.space.continuous.WrapAroundBorders(), 50,50);
11         GridFactory gridFactory = GridFactoryFinder.createGridFactory(null);
12         Grid<Object> grid = gridFactory.createGrid("grid", context,
13                     new GridBuilderParameters<Object>(new WrapAroundBorders(),
14                                 new SimpleGridAdder<Object>(), true, 50,50));
15         int zombieCount = 5;
16         for(int i = 0; i < zombieCount; i++){
17             context.add(new Zombie(space, grid));
18         }
19         int HumanCount = 100;
20         for(int i = 0; i < HumanCount; i++){
21             int energy = RandomHelper.nextIntFromTo(4, 10);
22             context.add(new Human(space, grid, energy));
23         }
24
25         for(Object obj: context){
26             NdPoint pt = space.getLocation(obj);
27             grid.moveTo(obj, (int) pt.getX(), (int) pt.getY());
28         }
29         return context;
30     }
31 }
```

Fonte: Adaptado de Collier e North (2016).

Desenvolvida a classe, agora é necessário ir no editor XML chamado *context.xml* adicionar o contexto com o mesmo id dado dentro no método *build()* da classe *JZombiesBuilder* e as projeções que foram criadas (*Grid*, *ContinuousSpace* e *network*), como mostrado na Figura 11. Essas projeções foram criadas no método *build()* e serão o ambiente da simulação.

Antes de iniciar a simulação, é necessário realizar algumas configurações: adicionar a classe inicializadora como *Data Loaders* e colocar os agentes no *Displays*. Para fazer isso é necessário configurar o *Scenario tree*, e para a execução usar o *context*, especificando o *data loader* (carregador de dados). Para iniciar as configurações, deve-se clicar na seta do botão *run* e selecionar *jzombie Model*, conforme mostrado na Figura 12.

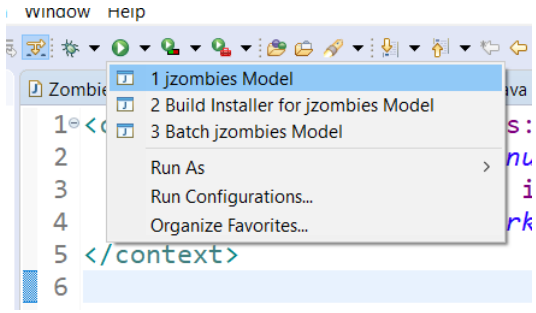
Ao clicar no *jzombies Model*, aparecerá a tela do Repast (Figura 13). No *Scenario Tree* (árvore de cenário) que apresenta o descritor do cenário, é clicado com o botão

Figura 11 – Classe *context* em XML

```
<context id="jzombies"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://repast.org/scenario/context">
  <projection type="continuous space" id="space" />
  <projection type="grid" id="grid"/>
   <projection type="network" id="infection network"/>
</context>
```

Fonte: Adaptado de Collier e North (2016).

Figura 12 – Execução do Jzombies Model



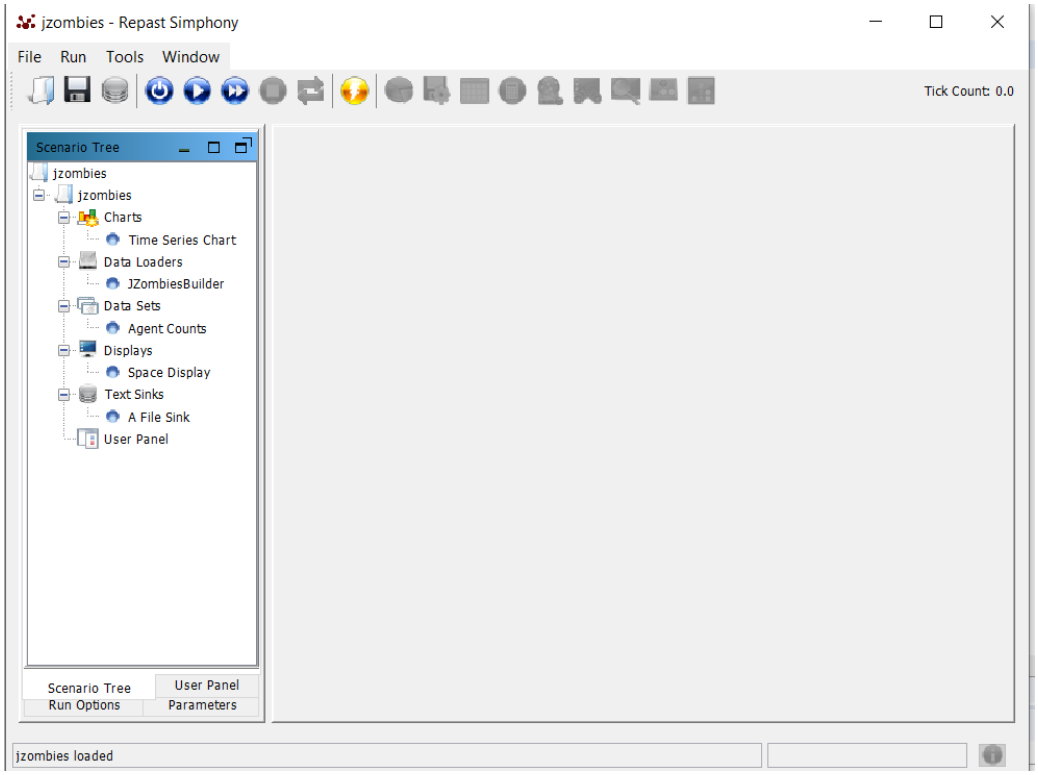
Fonte: Elaborado pela autora, 2020.

direito no *Data Loaders* e então no *Set Data Loader*. Abrirá uma janela *Select Data Source Type*, onde deve-se selecionar o *Custom ContextBuilder Implementation* e clicar em *Next*. No *combo box* da *Class Name* deverá aparecer o nome do *context* que criamos, o *jzombies.JZombiesBuilder*, então é clicado no botão *Finish*. Deverá aparecer o *JZombiesBuilder* como nome do *Data Loader*.

Em seguida deve-se criar um *display* simples (janela de exibição). No *Scenario Tree* basta clicar com o botão direito sobre o *Display* e então em *Add Display*. No diálogo de configuração do *Display*, digita-se o nome *Space Display*, deixando 2D como tipo. É selecionado nosso *space* e *infection network* como as projeções que queremos exibir. Para fazer isso, é necessário selecionar o space no *Projection and Value Layers* e clicar na flecha verde, deverá ser feita a mesma coisa para o *infection network*. A projeção na direita será aquela que é exibida enquanto que a da esquerda terão as possíveis projeções a serem exibidas. Com a(s) projeção(ões) escolhida(s), deverá ser clicado em *Next* (Figura 14).

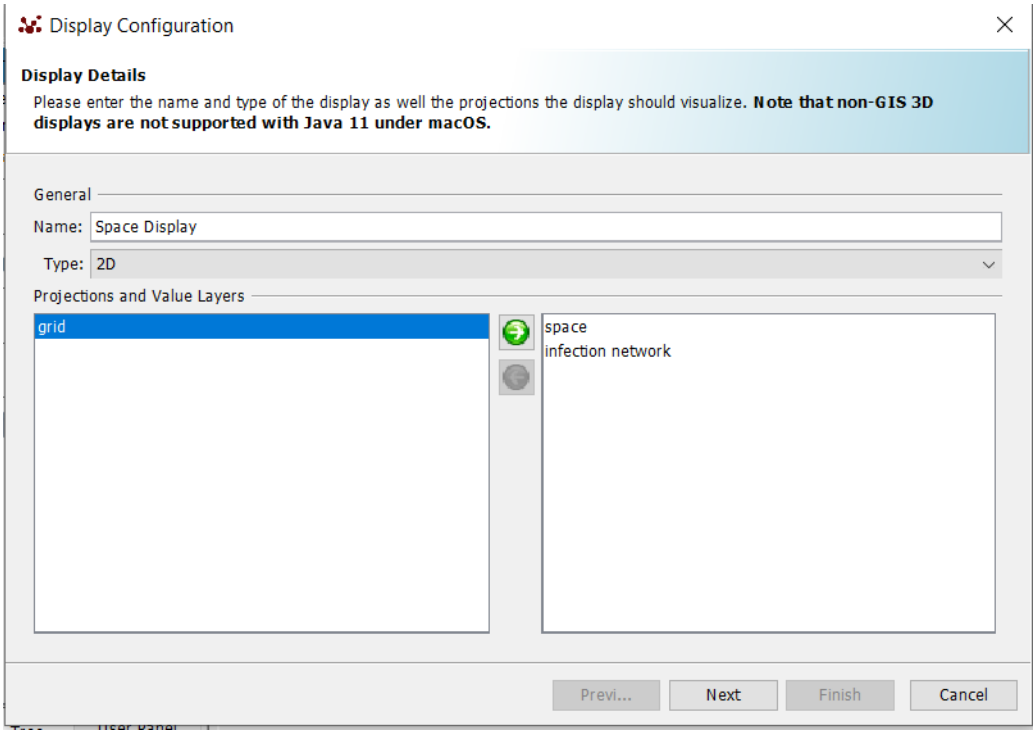
Na próxima janela, é feita a escolha dos agentes que serão exibidos no display. Foram selecionados os agentes *Humans* e *Zombie* para serem exibidos e clicado em *Next* (Figura 15), para então configurar como os agentes irão aparecer na simulação, como a forma (quadrado, redondo, etc) e a cor (verde, vermelho, azul, etc). Após escolhido a configuração dos agentes e clicado em *next* (Figura 16), também é possível configurar a *infection network*, depois é finalizado as configurações do *Display*.

Figura 13 – Tela de exibição do Repast com todas configurações já definidas



Fonte: Elaborado pela autora, 2020.

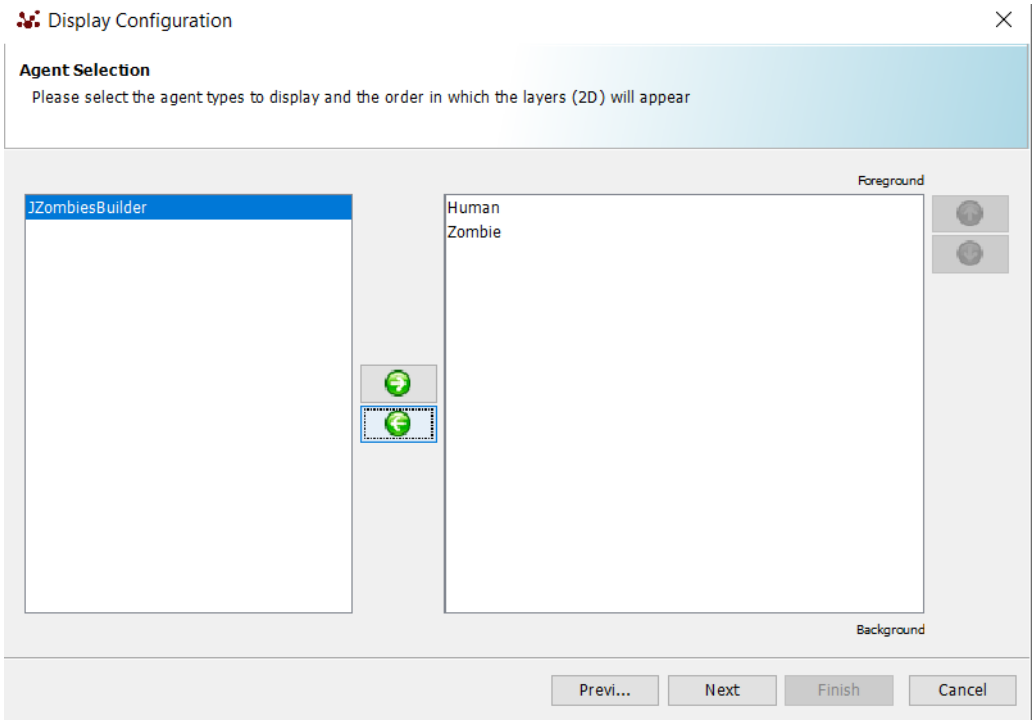
Figura 14 – ConFigurações do Display



Fonte: Elaborado pela autora, 2020.

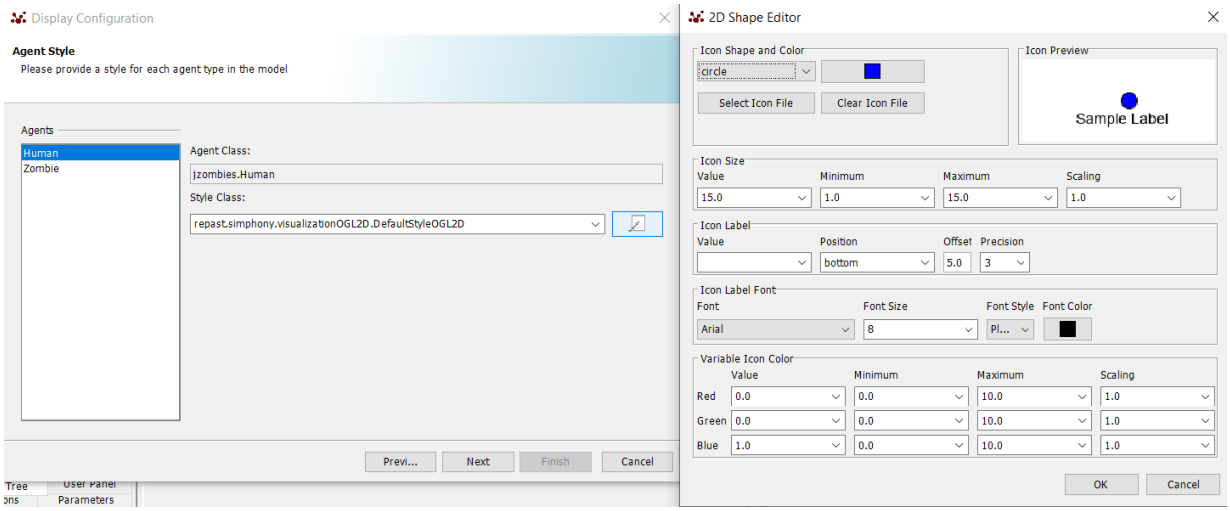
Ao rodar a simulação, será possível observar os *zombies* se aproximando dos *humans* e estes "correndo". Porém chega um momento em que os *humans* ficam sem energia

Figura 15 – Seleção dos agentes a serem exibidos no *display*



Fonte: Elaborado pela autora, 2020.

Figura 16 – ConFigurando o estilo dos agentes

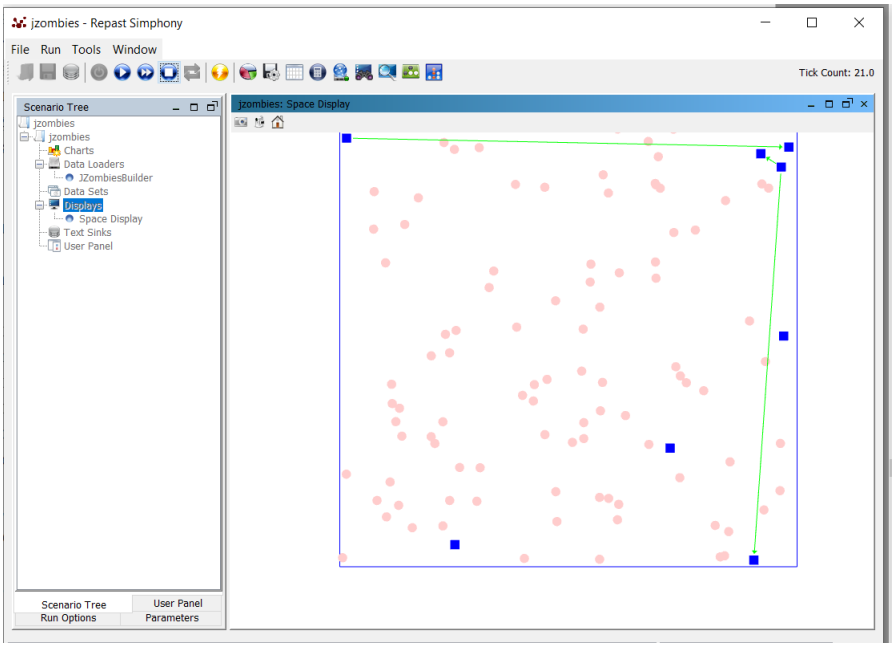


Fonte: Elaborado pela autora, 2020.

e acabam sendo infectados. Nesse momento poderemos observar que os *humans* serão substituídos por *zombie* e uma linha será estabelecida entre o novo *zombie* e o *zombie* que o infectou. Como mostrado na Figura 17, pode ser observado os *humans* como círculos de cor salmão e os *zombies* em quadrados azuis, aqueles *zombies* descendentes estão ligados de seus pais por uma flecha verde.

O Repast, através do seu *Scenario Tree*, permite realizar algumas ações, como por exemplo, gravar e mostrar algum tipo de dado, como o número de *zombies* e *humans* a

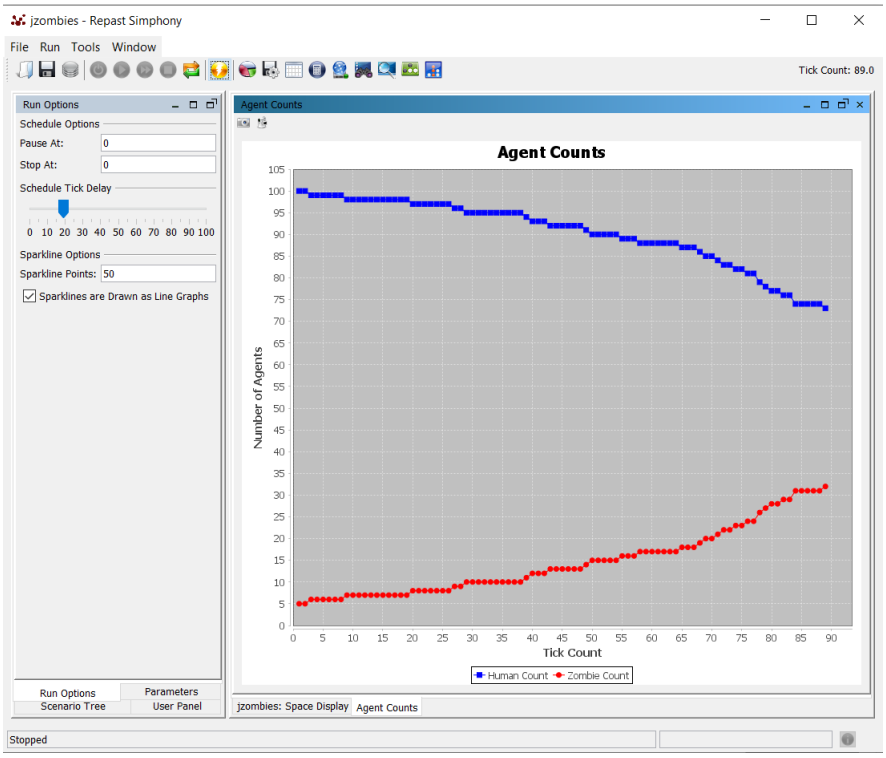
Figura 17 – Simulação zumbi com o network



Fonte: Elaborado pela autora, 2020.

cada *timestep*, através da adição de um *Data Set* dentro da simulação do Repast. Outra função é escrever esses dados clicando no *Text Sinks*, podendo ser no console ou em arquivo e até mesmo montar gráficos (Figura 18), clicando no *Charts*.

Figura 18 – Gráfico de contagem de agentes

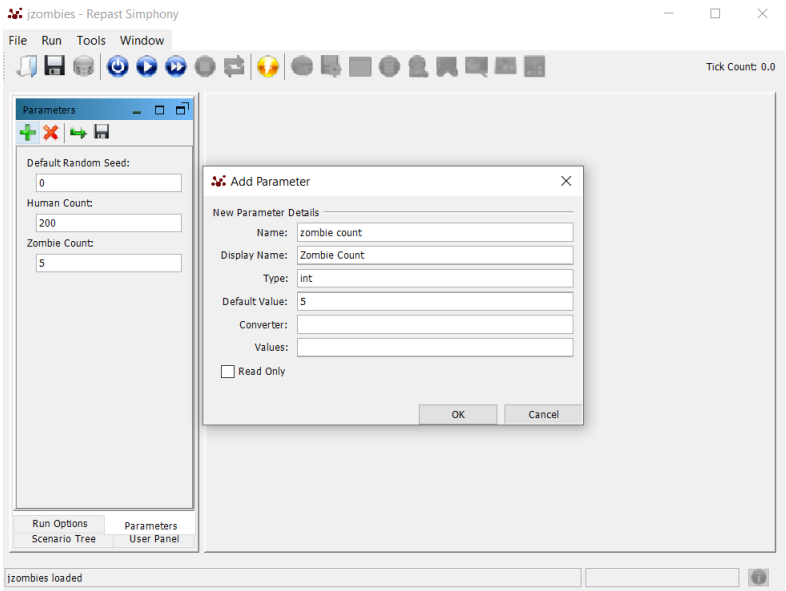


Fonte: Elaborado pela autora, 2020.

No presente exemplo, foi adicionado o número de *zombies* e *humans* diretamente no código. Porém é possível fazer com que o usuário escolha o número de *zombies* e *humans*. Na tela do Repast vem com a opção do painel *Scenario Tree* selecionada por padrão, na parte de baixo desse painel tem a opção "*Parameters*", onde é possível adicionar novos parâmetros para o usuário digitar.

Ao clicar no *Parameters*, aparecerá um botão de adicionar, ao clicar nele, será configurado o novo parâmetro a ser adicionado (Figura 19). Para ter acesso a esses parâmetros no código, a Figura 20 mostra um exemplo da recuperação do parâmetro *zombie_count*, que deverá ser feita na classe inicializadora *JZombiesBuilder*.

Figura 19 – Adicionando um novo parâmetro para adicionar a quantidade de *zombies* desejado



Fonte: Elaborado pela autora, 2020.

Figura 20 – Recuperação de parâmetro no código fonte

```
1  Parameters params = RunEnvironment.getInstance().getParameters();
2  int zombieCount = params.getInteger("zombie_count");
```

Fonte: Adaptado de Collier e North (2016).

2.4 XPAND

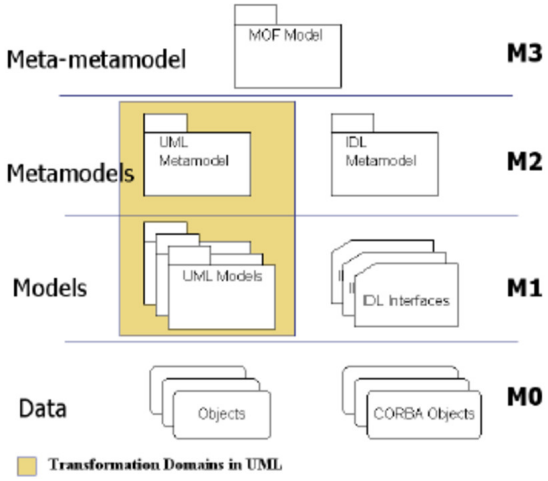
Xpand é uma *engine* (plataforma de desenvolvimento) para definição de *templates* textuais, sendo possível gerar qualquer tipo de documento textual, consequentemente qualquer tipo de linguagem de programação (SEVERIEN, 2008). Para a geração de código os dois passos básicos consistem em: (1) definir a estrutura do metamodelo e (2)

definir um ou mais *templates* que especificam como o gerador de código vai traduzir o modelo em código (FRIESE, 2010).

É necessário entender algumas definições dos conceitos relacionados ao desenvolvimento do gerador de código. A Figura 21, onde exemplifica as duas aplicações de MDD do lado esquerdo da mesma, tem as camadas da UML e do lado direito as camadas que descrevem as estruturas de dados e interface. Na camada M0 chamada de *data* (dados), encontra-se o sistema que é composto por objetos. Antes de construir um objeto é necessário ter um modelo do sistema (camada M1) sendo esse modelo composto por classes que o sistema deverá possuir, como pessoa, jogo, etc.

Para montar esse modelo, temos que saber quais elementos estão disponíveis para criar as classes e esses elementos são definidos na camada M2 de metamodelo. Segundo Friese (2010) a estrutura que um modelo deve processar é chamado de metamodelo.

Figura 21 – Arquitetura de uma abordagem MDD

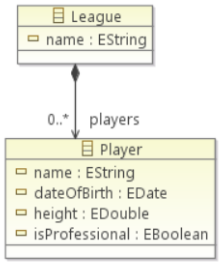


Fonte: Gallardo et al. (2005)

A Figura 22 apresenta um exemplo de metamodelo. Nessa Figura são definidos os metaelementos que estão disponíveis para desenvolver o modelo de um sistema de uma Liga de Boliche, que possui como atributos objetos do tipo Jogador. Esses metaelementos (Liga e Jogador) tem a possibilidade de possuir atributos. Dessa forma o metamodelo está definindo as regras, ou seja, quais elementos que podem existir no modelo. A partir do modelo, é possível instanciar o sistema com os objetos definidos no modelo e seus atributos.

O metamodelo da abordagem MDD4ABMS é formado por elementos que podem estar presentes em simulações, como entidade, agentes e ambientes. Uma entidade é um objeto que existe na simulação e um agente é uma entidade específica, como por exemplo

Figura 22 – Metamodelo da Liga de Boliche



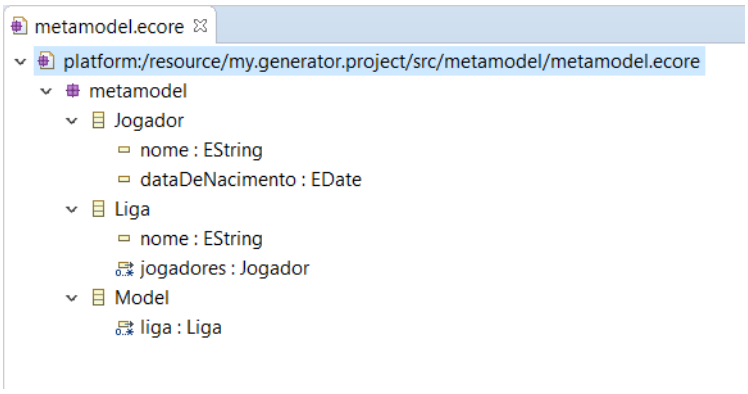
Fonte: Nogueira, Souza e Cortés (2008)

uma casa é uma entidade e um pitbull seria um agente. O agente pode conter habilidades, como a mobilidade.

As regras para geração de código em Xpand, segundo Friese (2010), levam em consideração os metaelementos contidos dentro do metamodelo, pois esses metaelementos definem as propriedades de cada elemento que o código irá gerar.

O exemplo de metamodelo da Figura 22 desenvolvido em Ecore (inclui um metamodelo para descrever modelos e suporte de tempo de execução para os modelos) pode ser visto na Figura 23. Nesse metamodelo são definidos os metaelementos (Jogador e Liga) e seus atributos que existirão na camada de modelo. Como no diagrama possui um relacionamento de jogador e liga, aqui vai ter um atributo do tipo jogador dentro de liga.

Figura 23 – Exemplo de metamodelo

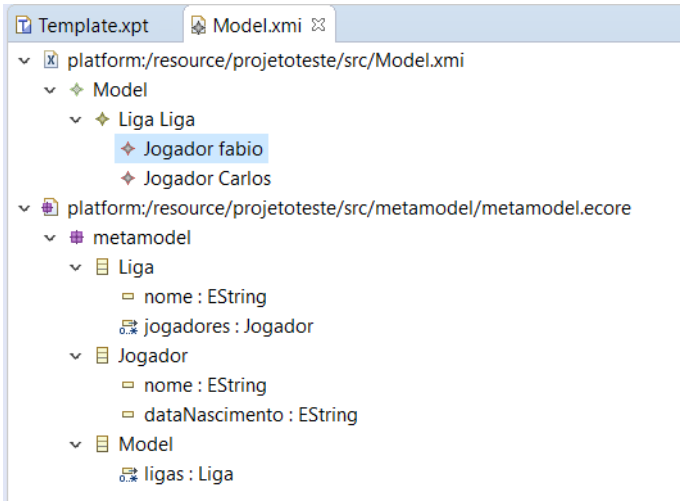


Fonte: Elaborado pela autora, 2020.

Na Figura 24 temos um exemplo de modelo XMI do metamodelo da Figura anterior. Nesse modelo temos o metaelemento liga que possui dois metaelementos jogadores.

As regras para geração de código são editadas dentro de um *template*. Nesse *template* deve ser importado o metamodelo para o gerador e o editor saber da estrutura do modelo, e as *tags Define* e *Enddefine* que abrem e fecham uma estrutura de *template*, respectivamente.

Figura 24 – Modelo baseado no metamodelo



Fonte: Elaborado pela autora, 2020.

A Figura 25 mostra uma estrutura Xpand de definição de campos dos metaelementos presentes na Figura 23. Será gerado código no formato HTML. É iniciado com a importação do metamodelo para o gerador e o editor saberem da estrutura do modelo.

Nesse exemplo da Figura 25 é definido um *code template* (modelo de código) chamado *main* ligado ao elemento do modelo do tipo *Model* (linha 2). Após é chamado outro *template* denominado liga, com a coleção de Jogadores (linha 3) e então o *template* é fechado (linha 4).

Na linha 6, é definido outro *template* chamado *liga* com o tipo em que está vinculado *Liga*. Foi usada a instrução *file* (linha 7) para especificar o arquivo no qual a saída será gravada, seu nome será derivado da concatenação do atributo *nome* da atual *Liga* com a *string* literal *.html*.

Nas linhas seguintes (8 a 32) desse *template* são estruturas de códigos HTML, com acessos aos atributos da atual Liga, como por exemplo nas linhas 11 e 18 para serem inseridos dentro do *template*. Na linha 25 temos um *expand*, que invocará outro *subtemplate* com o nome *jogador*. Esse *subtemplate* será chamado para todos os elementos *jogadores* da *Liga* atual.

Na linha 36 é criado o *template* jogador que está ligado ao elemento Jogador, onde retornará um pedaço de código HTML para criação de uma linha com duas colunas da tabela, onde o valor de cada coluna será o nome e a data de nascimento do atual “Jogador”.

Figura 25 – Código Xpand para geração de código a partir dos metaelementos do metamodelo

```
1  «IMPORT metamodel»
2  «DEFINE main FOR Model»
3  «EXPAND liga FOREACH Liga»
4  «ENDDEFINE»
5
6  «DEFINE liga FOR Liga»
7  «File nome ".html"»
8  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
9  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
10 <html xmlns="http://www.w3.org/1999/xhtml">
11 <head>
12   <title>«this.nome»</title>
13   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
14   <link rel="stylesheet" type="text/css" href="../static/style.css" />
15 </head>
16
17 <body>
18   <div id="page-wrap">
19     <h1>«this.nome»</h1><br/><br/>
20     <div id="tabela-jogadores">
21      <table>
22         <tr>
23             <th>Nome</th>
24             <th>Data de nascimento</th>
25         </tr>
26         «EXPAND jogador FOREACH this.jogadores»
27         </table>
28       <div style="clear: both;"></div>
29     </div>
30   </div>
31 </body>
32 </html>
33 «ENDFILE»
34 «ENDDEFINE»
35
36 «DEFINE jogador FOR Jogador»
37 <tr>
38     <td>«this.nome»</td>
39     <td>«this.dataDeNacimento»</td>
40 </tr>
41 «ENDDEFINE»
```

Fonte: Elaborado pela autora, 2020.

2.5 TRABALHOS CORRELATOS

Nesta seção serão apresentados trabalhos correlatos com a mesma finalidade do presente trabalho.

2.5.1 Extensão da Abordagem de Desenvolvimento Dirigido a Modelos Para Simulações com Agentes (MDD4ABMS) Para Suportar A Plataforma Repast (2019)

No trabalho desenvolvido por Tenfen (2019) foi utilizada a abordagem MDD4ABMS para desenvolver a portabilidade para a plataforma de simulação com agentes Repast. Para o desenvolvimento da portabilidade, segundo Tenfen (2019), foram realizadas al-

gumas etapas em ordem cronológica: Primeira etapa foram listados os elementos que estavam disponíveis para modelagem na ferramenta ABStractme; depois na segunda, foram criadas funcionalidades correspondentes da ferramenta ABStractme com a biblioteca de objetos Java da plataforma Repast, na terceira foi composta as dependências necessárias da Repast à ferramenta ABStractme e por último foram mapeadas as funcionalidades da ABStractme e da Repast para gerar os códigos fontes utilizando a linguagem Xpand.

A implementação foi realizada no Eclipse IDE e utilizou o plugin da ferramenta ABStractme para a modelagem. Para desenvolver um gerador de código para a plataforma Repast foi utilizada a linguagem Xpand.

O gerador desenvolvido suporta apenas ambientes simulados do tipo *grid* de duas dimensões, não tratando outros tipos de ambientes. As habilidades em que o trabalho se concentrou foram nas habilidades de mobilidade, sobrevivência e as definidas externamente, porém não foram desenvolvidas outras habilidades do agente, como é o caso da proposta desse trabalho que é o modelo de propagação de doenças.

2.5.2 EasyABMS: A domain-expert oriented methodology for agent-based modeling and simulation (2010)

O EasyABMS é um framework para simulação baseado em agentes que está conectado com a plataforma Repast. Segue uma abordagem dirigida a modelos e usa diagramas UML nas fases de análise, conceitual e de design do desenvolvimento do modelo (GARRO; RUSSO, 2010).

Esse framework foi especificamente concebido para a modelagem e simulação baseada em agentes de sistemas complexos e foi proposta para superar a falta de metodologias integradas capazes de orientar de forma efetiva desde a análise do sistema até a sua modelagem e análise de resultados da simulação (GARRO; RUSSO, 2010). Segundo Garro e Russo (2008), a metodologia EasyABMS define um processo iterativo para ABMS, que contempla etapas de análise do sistema, modelagem conceitual, design da simulação, geração do código da simulação, configuração da simulação e execução da simulação e análise dos resultados.

2.5.3 Análise dos trabalhos correlatos

Na Tabela 2 temos a comparação da portabilidade dos elementos da simulação para a plataforma respast dos trabalhos de Tenfen (2019), Garro e Russo (2010) e o presente trabalho (Petersen, 2021). Podemos analisar que a EasyABMS, desenvolvido por Garro

e Russo (2010), faz a portabilidade de modelos de agentes para a plataforma Repast assim como o trabalho desenvolvido por Tenfen (2019). A portabilidade das habilidades de mobilidade e sobrevivência e o ambiente *grid* de duas dimensões também são desenvolvidos por Tenfen (2019). Diferente dos outros dois autores, esse trabalho desenvolve a portabilidade do modelo de propagação de doença.

O presente trabalho e o de Tenfen (2019) utilizam a abordagem MDD4ABMS de Santos (2019) para realizar a portabilidade. A MDD4ABMS foi desenvolvida para fazer a portabilidade para a plataforma de simulação NetLogo e, com o trabalho realizado por Tenfen (2019), está fazendo para a plataforma de simulação Repast enquanto que o EasyABMS tem restrição de portabilidade, fazendo apenas para a plataforma de simulação Repast.

Tabela 2 – Análise dos trabalhos correlatos

	Tenfen (2019)	**Garro e Russo (2010)**	**Petersen (2021)**
Modelagem do agente	x	x	
Habilidade de mobilidade	x		
Habilidade de sobrevivência	x		
Ambiente *grid* de duas dimensões	x		
Habilidade do modelo de propagação de doença			x

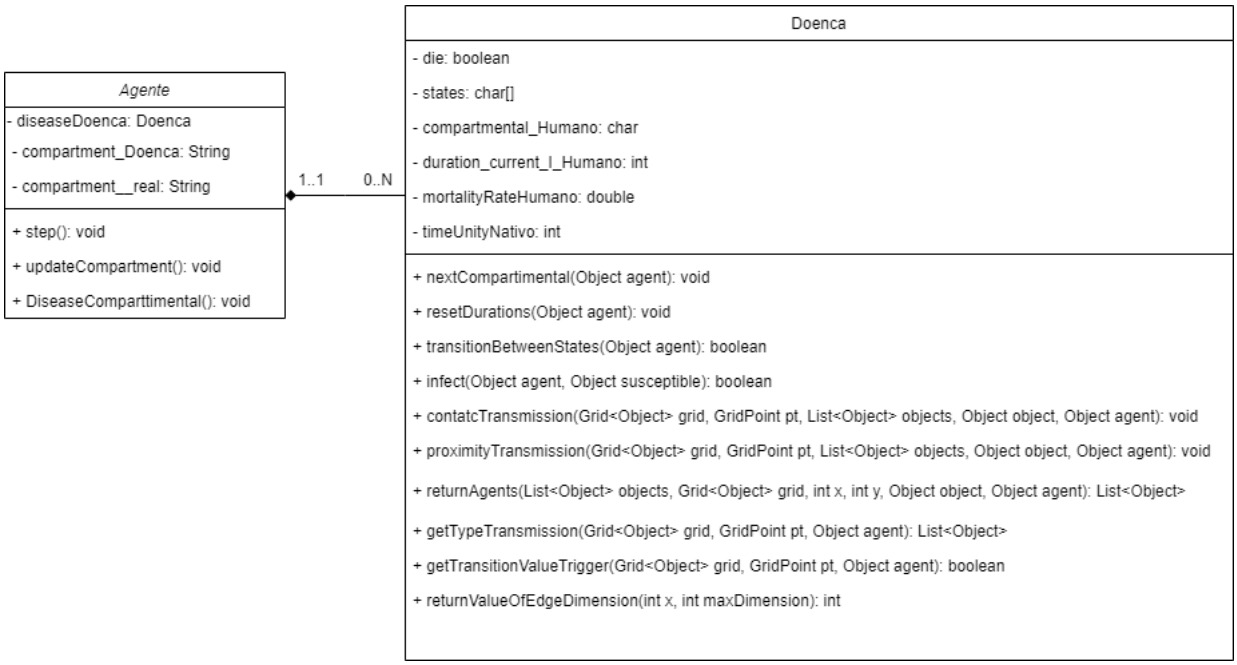
Fonte: Elaborado pela autora.

3 DESENVOLVIMENTO DA INOVAÇÃO

A portabilidade da geração de código para a plataforma de simulação com agentes Repast possui como intuito facilitar a vida do desenvolvedor de simulações com agentes. Portanto, esse trabalho se propôs a desenvolver a portabilidade da habilidade de propagação de doenças para a plataforma Repast através da ferramenta MDD4ABMS utilizando a linguagem Xpand para gerar códigos Java.

Para desenvolver o código Xpand foi necessário identificar o modelo da simulação gerado pela ferramenta ABStractme de acordo com as configurações do diagrama, bem como os elementos, funções e comandos disponíveis na plataforma Repast. Como a linguagem da plataforma Repast é Java e a mesma suporta a orientação a objetos, optou-se por desenvolver a doença como uma classe, formando um relacionamento de composição com o agente (Figura 26), onde um agente pode ter uma ou mais doenças porém a doença não existe sem um agente.

Figura 26 – Diagrama de Classe do relacionamento entre um agente e uma doença

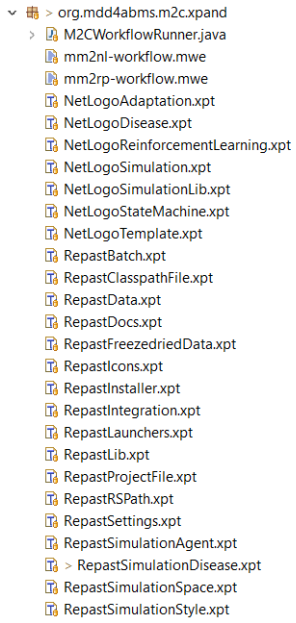


Fonte: Elaborado pela autora.

No diagrama, aparecem os atributos e métodos do agente que estão relacionados com as doenças e a geração do gráfico de saída, os demais atributos e métodos não pertinentes à doença e ao gráfico não são mostrados. Na Figura 27 temos todos os arquivos Xpand para a geração dos códigos NetLogo e Repast, incluindo aqueles arquivos desenvolvidos por Tenfen (2019). Nesse trabalho foi criado apenas o arquivo *RepastSimulationDise-*

ase.xpt. Porém foram adicionados novos códigos nos arquivos *RepastSimulationAgent.xpt*, *RepastSimulationSpace.xpt* e *RepastSimulationStyle.xpt*.

Figura 27 – Arquivos XPT para geração de códigos



Fonte: Elaborado pela autora.

A seguir são explicadas e apresentadas as principais etapas do desenvolvimento do gerador de código para a propagação de doenças. Na Seção 3.1 serão detalhados os atributos e métodos do agente enquanto que na seção 3.2 os da doença. Nos apêndices A e B temos exemplos de código Java gerado para as classes da doença e do agente, respectivamente.

3.1 AGENTE

No gerador de código do agente, desenvolvido por Tenfen (2019) é mostrado na Figura 28. Foi necessário realizar pequenas alterações em seu código para acrescentar a habilidade de propagação de doenças. As linhas destacas em amarelo claro são *templates* criados por Tenfen (2019). As destacas em laranja também são *templates* criados por Tenfen (2019) e que foram modificadas neste trabalho. Por fim, as linhas de código sem nenhuma cor correspondem aos *templates* novos, criados neste trabalho. Este esquema de cores é adotado em todos os trechos de código daqui por diante. Na linha 5, que é responsável por instanciar os atributos, foram acrescentados os atributos da doença e outros com a finalidade de gerar o gráfico de saída. Essas alterações serão explicadas na seção 3.1.1.

Figura 28 – Código em Xpand para definir a classe do agente

```
1  «DEFINE AgentClasses (mm::Agent agent) FOR mm::AgentBasedSimulation-»
2  package «GET_PACKAGE_NAME(this)»;
3  «EXpand importsAgentClass ((Agent) agent) FOR this»
4  public class «EXpand getClassName ((Agent) agent) FOR this-»  extends DefaultStyleOGL2D {
5          «EXpand agentAttributes ((Agent) agent) FOR this-»
6          «EXpand agentConstructor ((Agent) agent) FOR this-»
7          «EXpand agentGettersSetters ((Agent) agent) FOR this-»
8          «EXpand agentStep ((Agent) agent) FOR this-»
9          «EXpand agentDie ((Agent) agent) FOR this-»
10        «EXpand updateCompartment ((Agent) agent) FOR this-»
11         «EXpand randomAgentAtributte ((Agent) agent) FOR this-»
12         «EXpand createAgentCapabilities ((Agent) agent) FOR this-»
13         «EXpand bestSpotClass ((Agent) agent) FOR this-»
14         «EXpand getContentagentexternalCapability ((Agent) agent) FOR this-»
15        «EXpand DiseaseCompartimental ((Agent) agent) FOR this-»
16         «EXpand outPutDataSetMethods ((Agent) agent) FOR this-»
17 }
18 «ENDDEFINE»
```

Fonte: Adaptado de Tenfen (2019).

Nas linhas 6 e 7, que correspondem aos métodos do construtor e dos *getters* e *setters*, foram acrescentados, respectivamente, a instanciação da doença e acrescentado o *get* da doença, este último para retornar o objeto da doença. Os códigos das linhas 10 e 15 foram adicionados para, respectivamente, atualizar o compartimento e realizar a transmissão e progressão da doença. O método da linha 15, que é o principal método do gerador de código para gerenciar a evolução da doença, será explicado na seção 3.1.2.

No *step* (linha 8) acrescentou-se a chamada do novo método Java gerado pelo Xpand da linha 15, para a cada *tick* da simulação poder fazer a transmissão e/ou a progressão da doença. Nos métodos de *output* (linha 16) foi acrescentado o método de verificação do compartimento da doença para poder realizar a contagem dos agentes, que serão explicados na seção 3.1.3.

3.1.1 Atributos da Doença

O agente que possuir uma doença terá atributos que estão relacionados com a mesma, conforme mostrado na Figura 29. Quando a geração de códigos é feita, o nome da doença e do agente é substituído pelo configurado no modelo da simulação, por exemplo, se a doença se chamar Sarampo, terá um atributo *diseaseSarampo*, a mesma coisa irá acontecer com os atributos da linha 2, onde será trocado o nome *Doenca* por *Sarampo*. Os atributos da linha 2 são para fazer a verificação do compartimento da doença. Caso

o agente não possua uma saída, eles não serão gerados. Esse assunto será abordado na seção 3.1.3.

Figura 29 – Atributos da doença Sarampo no agente

```
1 private Doenca diseaseDoenca;
2 private String compartment_Doenca, compartment_Doenca_real;
```

Fonte: Elaborado pela autora.

Para gerar os atributos da doença, foi necessário acrescentar um trecho de código Xpand no *template* de geração dos atributos dos agentes (linha 5 da Figura 26) criado por Tenfen (2019), conforme mostrado na Figura 30. O *FOREACH* da linha 1, o *IF* da linha 2 e o código das linhas 10, 11, 43 e 44 já existiam. A linha 1 percorre todos os atributos do agente, enquanto que a linha 2 faz uma verificação se o *primitiveType* do atributo é diferente de *undefined*.

Foram ascrescentadas todas as verificações para gerar o código das linhas 11, 23 e 24. A variável *agentsWithDiseaseModel* (linha 4) armazenará todos os agentes que possuirem a doença e caso ela não esteja vazia, na linha 6 é feito um *FOREACH*, que irá percorrer todos agentes, que um por um serão atribuídos a variável *diseaseModel* (linha 7). Caso possuir uma doença, o código da linha 13 é gerado, se não, o código original será gerado (linhas 10 ou 30).

Se possuir um gráfico de saída, o código das linhas 25 e 26 também será gerado. Das linhas 14 até 24 são verificados se foi configurado um gráfico de saída para aquele agente. Caso não seja da mesma classe do agente (linha 8), o próximo agente dentro da variável *diseaseModel* é chamado para fazer as verificações. Um exemplo de código gerado pelas linhas 13, 25 e 26 é aquele mostrado anteriormente na Figura 29.

O atributo da doença é instanciado no construtor da classe do agente e o atributo *String* do compartimento da doença é atualizado no método *updateCompartment()*, mostrado na Figura 31.

O método *updateCompartment()* é chamado sempre que a doença muda de compartimento. Dessa forma ele verifica o compartimento atual da doença em que o agente se encontra, para então atualizar os atributos do agente. Por exemplo, a doença estava no compartimento S (suscetível) e o agente acabou sendo infectado, então a doença irá para o compartimento I (infectado) e ao entrar nesse método, irá para o *case* 'I' e atualizará o atributo *int* infectado para 1, os demais para 0 e a *String* correspondente ao compartimento para 'I'. Desta forma a cada *tick* da simulação, a quantidade de agente em cada compartimento será atualizado.

Figura 30 – Código Xpand para a geração dos atributos do agente

```
«FOREACH agent.attributes AS attr»
 «attr.primitiveType.toString() !=''UNDEFINED''»
  «LET (List[mm::Agent]) this.concerns.entities.select( a | mm::Agent.isInstance(a) &&
 ((mm::Agent)a).capabilities.exists( c | mm::DiseaseModel.isInstance((c)) )) AS agentsWithDiseaseModel»
 «IF ! agentsWithDiseaseModel.isEmpty»
  «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select(c |
  mm::DiseaseModel.isInstance(c)) AS diseaseModel»
   «IF diseaseModel.agent.name == CAPTALIZE(agent.name.toString()) »
    «IF attr.name != "compartment_"+diseaseModel.id »
     private «EXpand getAttributeType FOR attr»«attr.name» =
          «GET_ATTRIBUTE_DEFAULT_VAL(attr.primitiveType.toString())»;
    «ELSE»
     private «diseaseModel.id» disease«diseaseModel.id»;
     «LET this.concerns.select(a | a.outputDatasets.size > 0) AS concerns-»
     «FOREACH concerns AS concernOutPut»
      «LET (mm::Concern) concernOutPut AS concern»
      «FOREACH concern.outputDatasets AS outputDataSet»
       «FOREACH outputDataSet.outputData AS outPutData»
        «IF outPutData.name == outputDataSet.outputData.get(0).name»
         «IF outPutData.entity.name.toString().toLowerCase().trim() ==
         agent.name.toString().toLowerCase().trim()»
          «IF outPutData.metaType == Aggregation»
          «LET (mm::Aggregation) outPutData AS aggregation»
           «IF aggregation.aggregationType.toString().toUpperCase() == "COUNT"»
            «IF aggregation.name != "UNDEFINED"»
              private String compartment_«diseaseModel.id»,
                compartment_«diseaseModel.id»_real;
            «ENDIF»
           «ENDIF»
          «ENDLET»
         «ENDIF»
        «ENDIF»
       «ENDIF»
      «ENDFOREACH»
     «ENDFOREACH»
     «ENDLET»
    «ENDFOREACH»
    «ENDLET»
   «ENDIF»
  «ENDIF»
 «ENDFOREACH»
 «ELSE»
    private «EXpand getAttributeType FOR attr»«attr.name» =
         «GET_ATTRIBUTE_DEFAULT_VAL(attr.primitiveType.toString())»;
  «ENDIF»
 «ENDLET»
 «ENDIF»
«ENDFOREACH»
```

Fonte: Elaborado pela autora.

Figura 31 – Método UpdateCompartment em Java

```
public void updateCompartment() {
   switch (this.getDiseaseDoenca().getCompartmental()) {
         case 'S' :
                  this.compartment_Doenca_real = "S";
                  break;
         case 'E' :
                  this.compartment_Doenca_real = "E";
                  break;
         case 'I' :
                  this.compartment_Doenca_real = "I";
                  break;
         case 'R' :
                  this.compartment_Doenca_real = "R";
                  break;
         default :
                  this.compartment_Doenca_real = "";
                  break;
   }
}
```

Fonte: Elaborado pela autora.

3.1.2 Método de gerenciamento da propagação da doença

Em cada *tick* da simulação, o método *step()* do agente é chamado e aqueles que possuem a doença chamam o método *DiseaseCompartimental()*, que só irá aparecer caso

o agente possua a habilidade de propagação da doença. O código Xpand para a geração do método *DiseaseCompartimental()* pode ser visto na Figura 32.

Figura 32 – Código Xpand para a geração do método de gerenciamento da propagação da doença na classe do agente

```
1  «DEFINE DiseaseCompartimental (mm::Agent agent) FOR mm::AgentBasedSimulation»
2  «LET (List[mm::Agent]) this.concerns.entities.select( a | mm::Agent.isInstance(a) &&
3  ((mm::Agent)a).capabilities.exists( c |
4  mm::DiseaseModel.isInstance((c)))) AS agentsWithDiseaseModel»
5  «IF ! agentsWithDiseaseModel.isEmpty»
6  public void DiseaseCompartimental(){
7    «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select(c |
8    mm::DiseaseModel.isInstance(c) && ((DiseaseModel)c).agent == agent) AS diseaseModel»
9    if(disease«diseaseModel.id».getCompartmental() == 'I') {
10     GridPoint pt = grid.getLocation(this);
11     //verificar se a transmissão é por contato ou proximidade
12     List<Object> Objects = this.disease«diseaseModel.id».getTypeTransmission(grid, pt, this);
13         if (Objects.size() > 0) {
14           for (Object obj : Objects) {
15           «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select( c |
16           mm::DiseaseModel.isInstance(c)) AS diseaseModelA»
17             if(obj instanceof «diseaseModelA.agent.name»){
18               «diseaseModelA.agent.name» agent = («diseaseModelA.agent.name») obj;
19               if(agent.getDisease«diseaseModel.id»().getCompartmental() == 'S'){
20                 if(disease«diseaseModel.id».infect(this, agent)){
21                   agent.getDisease«diseaseModel.id»().nextCompartimental();
22                   agent.updateCompartment();
23                 }
24               }
25           }
26          «ENDFOREACH»
27          }
28     }
29   }
30   if(this.disease«diseaseModel.id».getCompartmental() != this.disease«diseaseModel.id».getStatePosition(0)) {
31     if(this.disease«diseaseModel.id».transitionBetweenStates(this)){
32           die();
33         }
34   }
35   updateCompartment();
36   «ENDFOREACH»
37 }
38 «ENDIF»
39 «ENDLET»
40 «ENDDEFINE»
```

Fonte: Elaborado pela autora.

No método Java *DiseaseCompartimental()* que é gerado, o agente realiza a progressão da sua doença, se estiver infectado poderá infectar outros agentes e até mesmo a possibilidade da sua morte. Para realizar todas essas ações, são chamados métodos da classe da doença. Inicialmente é verificado se o compartimento do agente é o "I" (linha 9), caso não for é feito a progressão da doença (linha 31) com a possibilidade de sua morte (linha 32), caso ele não esteja no compartimento "S". Se o agente se encontrar no compartimento "I", será verificado todos os agentes próximos (linha 12), de acordo com as configurações de transmissão explicadas na seção 2.2, para então verificar a possibilidade de transmissão da doença (linha 20).

Na linha 9 é verificado se o agente está infectado para então guardar dentro da variável *Objects* todos os agentes que estão próximos (linha 12). É gerado um laço de repetição *FOR* Java na variável citada para verificar se o agente *obj* (linha 14) se encontra no compartimento suscetível (linha 19), caso esteja, é chamado o método *infect()* (linha 20) que verificará as condições para que o agente *obj* se torne infectado.

O *FOREACH* da linha 15 serve para repetir o código das linhas 17 a 25 para cada classe de agente que possuir a doença. Caso a simulação possua três tipos de agentes com a habilidade de propagação de doença, então serão gerados três *IFS* no código Java, um para cada tipo de agente. Desta forma a transmissão da doença se dará do agente tipo 1 com ele mesmo, dele com o agente tipo 2 ou dele com o agente tipo 3. A mesma coisa irá acontecer com as demais classes de agentes, por exemplo o agente tipo 2, a transmissão ao invés de ser do agente tipo 1 para o agente tipo 2, será do agente tipo 2 para o agente tipo 1.

Por último é chamado o método *transitionBetweenStates* da doença (linha 31), no qual é gerado o código Java que faz a transição de um compartimento para o outro. Esse método retornará *true* se as condições para a mortalidade forem favoráveis. Na seção 3.2.5 será detalhado esse método.

3.1.3 Saídas da Simulação

Nas saídas da simulação, originalmente implementadas por Tenfen (2019), foram necessários acrescentar os códigos para a geração do gráfico da habilidade de propagação de doença. Para a geração do gráfico, o programa faz uma contagem de agentes. Para isso, ele percorre todos agentes realizando uma soma dos retornos de cada método de *output*. Devido a isso, os métodos retornarão 0 ou 1, assim, se existem 20 agentes e 5 se encontram no estado infectado, apenas cinco agentes terão como retorno do método de *output* para o estado infectado o valor 1, os outros 15 agentes retornarão o valor 0.

Conforme já explicado, o método *updateCompartment()* da Figura 31 é responsável por atualizar o compartimento atual do agente. A *String* do compartimento da doença recebe a letra correspondente ao seu compartimento. O agente possui duas *Strings* como atributo, utilizando como exemplo a Figura 29, a *compartment_Doenca_real* sempre receberá o valor correspondente ao compartimento que o agente se encontra. Já a outra, a *compartment_Doenca*, seu valor será atribuido em cada método de geração do gráfico para ser comparada ao compartimento do agente.

Para cada saída do gráfico relacionados aos compartimentos do agente é chamado o método que contém o código Xpand da Figura 33. Os trechos de código em destaque amarelo claro são códigos desenvolvidos por Tenfen (2019). Este método gera um código Java que verifica se possui agentes com a doença e se existe um critério de elegibilidade, para então instanciá-lo (linha 5). Cada critério é gerado uma linha no gráfico, ou seja, se quer gerar uma saída para os agentes infectados, o critério de eligibilidade terá o valor,

seguindo o exemplo da Figura 29, *compartment_Doenca = "I"*. É com esse critério de eligibilidade que é feita a comparação com o atributo *compartment_Doenca_real*.

Figura 33 – Código Xpand para a saída da simulação

```
1  «LET (List[mm::Agent]) this.concerns.entities.select(a | mm::Agent.isInstance(a) &&
2  ((mm::Agent)a).capabilities.exists(c | mm::DiseaseModel.isInstance((c)))) AS agentsWithDiseaseModel»
3  «IF ! agentsWithDiseaseModel.isEmpty»
4    «IF aggregation.entityEligibilityCriteria != null»
5     this.«EXpand agentGetPeriodicityValue(aggregation.entityEligibilityCriteria) FOR this»;
6      «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select(c |
7      mm::DiseaseModel.isInstance(c) && ((DiseaseModel)c).agent == agent) AS diseaseModel»
8        if (((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() %
9        «EXpand agentGetPeriodicityValue(aggregation.periodicityValue) FOR this») == 0)
10       «IF aggregation.entityEligibilityCriteria != null» && (this.compartment_«diseaseModel.id» ==
11       this.compartment_«diseaseModel.id»_real)«ENDIF») {
12           return 1;
13        }
14        return 0;
15     «ENDFOREACH»
16   «ELSE»
17    return 1;
18   «ENDIF»
19 «ENDIF»
20 «ENDLET»
```

Fonte: Elaborado pela autora.

Nas linha 6 e 7 é feito um *FOREACH* para percorrer todos os agentes que possuem a doença para então verificar em qual *tick* a simulação se encontra e se o critério de elegibilidade está de acordo com o compartimento atual do agente (linhas 8 a 11) para então retornar 1 (linha 12) caso for verdadeiro ou 0 (linha 14) para falso. Como mencionado anteriormente, o programa faz uma contagem, ao retornar 1, o programa contabiliza mais um agente naquele compartimento.

Para melhor visualização do código Java gerado, temos um exemplo na Figura 34. Cada compartimento tem seu próprio método. Nesse exemplo é mostrado o método do compartimento suscetível. O critério de elegibilidade da linha 5 da Figura 33 gerou a linha 2 da Figura 34. Nas linhas 3 e 4, é feito a verificação do *tick* e do compartimento. Retorna 1 caso for verdadeiro (linha 5) e 0 caso for falso (linha 7).

Figura 34 – Exemplo de código Java gerado para geração da linha suscetível do gráfico

```
1  public int countsusceptiveis() {
2    this.compartment_Doenca = "S";
3     if (((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() % 1) == 0) &&
4    (this.compartment_Doenca == this.compartment_Doenca_real)) {
5           return 1;
6    }
7    return 0;
8  }
```

Fonte: Elaborado pela autora.

Como mencionado anteriormente, cada compartimento terá o seu próprio método. Dessa forma, se a doença possui o modelo compartimental SEIR, e todos geram saídas, então teremos quatro métodos de contagem, mudando o nome do método e o critério de

eligibilidade, onde terão, por exemplo, para o compartimento R, o nome de *coutrecuperados* e o critério *compartment_Doenca = "R"*.

3.2 DOENÇA

Seguindo o modelo criado pela ferramenta ABStractme, a estrutura Xpand que gerará o código Java da mesma segue as regras mostradas na Figura 35. Nesse código, são feitos todos os *imports* (linha 3), é criada a classe com o nome configurada no diagrama (linha 4), é definido todos os atributos (linha 5), realizado todos os *getters* e *setters* (linha 7), criado o construtor (linha 6) e todos os outros métodos que realizam a transmissão, progressão e a mortalidade da doença. A introdução da mesma foi desenvolvida na classe *Space* na hora em que se instanciam os agentes, que será detalhada na seção 3.2.3.

Figura 35 – Código em Xpand para definir a classe da Doença

```
1  «DEFINE DiseaseClasses (mm::DiseaseModel diseaseModel) FOR mm::AgentBasedSimulation-»
2  package «GET_PACKAGE_NAME(this)»;
3  «EXpand importsDiseaseClass ((DiseaseModel)diseaseModel) FOR this-»
4  public class «EXpand getClassName ((DiseaseModel)diseaseModel)  FOR this-» {
5          «EXpand diseaseAttributes ((DiseaseModel)diseaseModel) FOR this-»
6          «EXpand diseaseConstructor ((DiseaseModel)diseaseModel) FOR this-»
7          «EXpand diseaseGettersSetters ((DiseaseModel)diseaseModel) FOR this-»
8          «EXpand methodNext ((DiseaseModel)diseaseModel) FOR this-»
9          «EXpand resetDurations ((DiseaseModel)diseaseModel) FOR this-»
10         «EXpand transitionBetweenStates ((DiseaseModel)diseaseModel) FOR this-»
11         «EXpand infect ((DiseaseModel)diseaseModel) FOR this-»
12         «EXpand contactTransmission ((DiseaseModel)diseaseModel)  FOR this-»
13         «EXpand getTransmissionByInfected  FOR this-»
14         «EXpand returnAgents ((DiseaseModel)diseaseModel) FOR this-»
15         «EXpand proximityTransmission ((DiseaseModel)diseaseModel) FOR this-»
16         «EXpand returnValueOfEdgeDimension ((DiseaseModel)diseaseModel) FOR this-»
17         «EXpand getTypeTransmission FOR this-»
18         «EXpand transitionValue ((DiseaseModel)diseaseModel) FOR this-»
19 }
20 «ENDDEFINE»
```

Fonte: Elaborado pela autora.

Os métodos das linhas 8 a 9 estão relacionados à progressão da doença, neles são verificado qual o próximo compartimento que a doença deverá ir e resetar a duração do compartimento em que o agente se encontrava anteriormente. Esses métodos são chamados no método *transitionBetweenStates* da linha 10, que fará a verificação se o agente irá mudar de compartimento.

Para fazer a transmissão da doença, o agente chamará o método Java gerado pela linha 13, que verificará qual tipo de transmissão foi configurada para ele, se é por próximidade (linha 15) ou por contato (linha 12). Após encontrar os agentes que poderão ser infectados, chamará o método *infect()* (linha 11). Nas próximas subseções serão detalhadas cada etapa.

3.2.1 Atributos

Todas as doenças possuem atributos que são comuns, como um *char* para guardar o atual compartimento da doença, um *boolean* para indicar se o agente ainda está vivo e um *array* de *char* para guardar os compartimentos em que o agente pode passar, ou seja, o modelo de compartimentos da doença, como os modelos compartimentais SIR e SEIR.

Esses atributos são gerados diretamente no código sem necessitar de código Xpand. Os que dependerão das configurações possuem códigos Xpand para verificação do modelo da simulação. Na Figura 36 temos o código para a geração dos atributos que possuem progressão do tipo determinísticas. Por exemplo, no modelo da simulação possui em sua *ProgressionTransition* o valor de 10 para ir do compartimento I para o R, então o compartimento I terá um atributo de duração que será acrescentado a cada *tick* até chegar ao 10, para então mudar para o compartimento R. Os outros tipos de progressão não precisaram de atributos, sendo que suas condições serão geradas automaticamente no código, que serão detalhadas na seção 3.2.5.

Figura 36 – Código Xpand para a geração da duração dos compartimentos

```
1 «FOREACH states AS state ITERATOR iState»
2 «IF state.name != 'S'»
3         private int duration_current_«state.name» = 0;
4 «ENDIF»«ENDFOREACH»
```

Fonte: Elaborado pela autora.

Na Figura 37 temos a geração do código dos atributos relacionados a mortalidade. O código é iniciado fazendo um *FOREACH* em todos os *states* da doença (linha 1). Após, é feito outro *FOREACH* para percorrer todas as transições que os *states* possuem (linha 2). Na linha 3 é verificado se o alvo da transição é um pseudoestado *dead*, que significa que existe uma configuração de mortalidade, pois conforme explicado previamente na seção 2.2, as mortes causadas pela doença são especificadas como transições para um pseudoestado adicional chamado *dead* cujo seu tipo é *final*.

A mortalidade "ao sair do compartimento" é a única que o nome do seu *state* não é o nome do compartimento, desta forma foi necessário fazer a verificação (linha 4) para não utilizar o *«state.name»* (linha 8) ao declarar o compartimento I (linha 5). Caso a mortalidade for "em um momento específico após infectado", terá um parâmetro a mais que representa o momento específico (linha 10).

Figura 37 – Código Xpand para a geração dos atributos da mortalidade

```
1  «FOREACH diseaseModelAgent.states AS state»
2    «FOREACH state.outgoing.select( t | StandardTrigger.isInstance(t.trigger) ) AS transition»
3      «IF transition.target.name == "dead"»
4        «IF state.name == "whenleavingcompartmentI"»
5          private double mortalityRate«diseaseModelAgent.agent.name»I;
6          private int timeUnity«diseaseModelAgent.agent.name»;
7        «ELSE»
8          private double mortalityRate«diseaseModelAgent.agent.name»«state.name»;
9          «IF StateMachineStateTimeExpression.isInstance(transition.guard)»
10           private int timeUnity«diseaseModelAgent.agent.name»;
11         «ENDIF»
12       «ENDIF»
13     «ENDIF»
14   «ENDFOREACH»
15 «ENDFOREACH»
```

Fonte: Elaborado pela autora.

3.2.2 Construtor

É no construtor que os valores são atribuidos às suas variáveis. Na Figura 38 temos a atribuição dos valores das variáveis relacionadas à mortalidade. Quando é especificado o tipo da mortalidade de "ao sair do compartimento", acrescenta-se um *state* com nome de *whenleavingcompartmentI* com o alvo da *transition* para *dead*. Já para os tipos "a todo momento", "em um momento específico após infectado" e "quando a condição ocorrer", é acrescentado um *state* chamado "I" com o alvo da *transition* para *dead*.

Figura 38 – Código Xpand do construtor para atribuição dos valores as variáveis relacionados a mortalidade

```
1  «FOREACH diseaseModelAgent.states AS state»
2   «FOREACH state.outgoing.select( t | StandardTrigger.isInstance(t.trigger)) AS transition»
3    «IF transition.target.name == "dead"»
4     «IF state.name == "whenleavingcompartmentI"»
5     this.mortalityRate«diseaseModelAgent.agent.name»I =
6     «EXpand sourceValue(((StandardTrigger)transition.trigger).value) FOR diseaseModel.agent»;
7     «ELSE»
8      this.mortalityRate«agent.name»«state.name» =
9      «EXpand sourceValue(((StandardTrigger)transition.trigger).value) FOR diseaseModel.agent»;
10     «IF StateMachineStateTimeExpression.isInstance(transition.guard)»
11      this.timeUnity«agent.name» = «EXpand
12      sourceValue(((StateMachineStateTimeExpression)transition.guard).comparedValue) FOR diseaseModel.agent»;
13     «ENDIF»
14    «ENDIF»
15   «ENDIF»
16  «ENDFOREACH»
17 «ENDFOREACH»
```

Fonte: Elaborado pela autora.

O código é iniciado com dois *foreach*, para passar pelos *states* (linha 1) que a doença possui e pelas transições que os *states* possuem (linha 2). Na linha 3 é feita a verificação se o *state* está relacionado a alguma mortalidade. Na linha 4 é verificado se o tipo da mortalidade é "ao sair do compartimento". Na linha 10 verifica se possui uma *TimeExpression*, caso possua, é o tipo "em um momento específico após infectado".

O *TimeExpression* é adicionado a *guard* da transição para guardar o valor do tempo. Caso não possua o *TimeExpression*, é "a todo momento" ou "quando a condição ocorrer". Nas linhas 6, 9 e 11 é chamado o *template sourceValue* que é responsável por gerar o valor

de alguma fonte, que nesses casos seriam os valores da taxa de mortalidade e do tempo configurado no tipo “em um momento específico após infectado”.

3.2.3 Introdução

O modelo da introdução da doença possui os elementos da periodicidade, da quantidade e da seleção. A periodicidade poderá ser *aperiodic* ou *periodic*. No primeiro caso, os agentes infectantes serão introduzidos tudo de uma vez e somente naquele instante específico. No *periodic*, os agentes infectantes serão introduzidos após cada intervalo de tempo , ou seja, serão introduzidos periodicamente.

A quantidade poderá ser *deterministic* ou *probabilistic*. O primeiro é uma quantidade específica de agentes enquanto que o segundo será por probabilidade. A seleção poderá ser *arbitrary* ou *eligible*, o primeiro escolhe agentes de forma aleatório enquanto que o segundo é devido a algum critério. Abaixo serão detalhados os códigos onde os atributos do tipo *List* chamados *cadidateAgents* e *selectedAgents* são declarados no início do método, por isso suas declarações não apareceram neles.

A Figura 39 mostra o código Xpand para a quantidade *deterministic*. Nele é verificado o critério de seleção (linha 4). Caso for *arbitrary* gerará o código Java da linhas 6 a 37, se não, gerará os da linha 40 a 49 (*eligible*). Em ambos os casos será feito um *FOR* para percorrer todos os objetos do contexto e será verificado a instância de cada um dos objetos.

No *arbitrary* será verificado se o agente se encontra no compartimento suscetível (linhas 32 e 33) enquanto que no *eligible* além de verificar o compartimento, também terá o critério (linhas 44 e 45). Por último é gerado o código que, dentre os agentes candidatos, irá selecionar os agentes que se tornarão infectados (linhas 51 a 62).

No critério *arbitrary*, todos os agentes presentes na simulação serão atribuídos a variável *candidatesAgents* (linha 34), a declaração dessa variável não aparece no código. Se o critério for *eligible* apenas os agentes que satisfaçam a condição é que serão salvos dentro dessa variável (linha 46). Na linha 51 é feito a verificação se a *list* com os agentes candidatos é maior que a determinada na especificação da introdução da doença, para então selecionar a quantidade certa de agentes dentro da *list* de agentes candidatos e passar para a *list* com os candidatos que serão os selecionados, a *selectedAgents*. Caso a quantidade seja menos que a determinada, então todos agentes candidatos serão escolhidos (linha 61).

Caso a quantidade for *probabilistic* (Figura 40), também é feito uma verificação para saber o critério de seleção (linha 4). Se for *arbitrary* (linhas 6 a 13) é gerado o código

Figura 39 – Código Xpand da introdução dos agentes infectados no modo determinístico

```
1  «IF diseaseModel.introduction.quantityType == mm::IntroductedQuantityType::DETERMINISTIC»
2  //deterministic
3  //arbitrary or eligible?
4   «IF diseaseModel.introduction.selectionCriteria == mm::IntroductionSelectionCriteria::ARBITRARY_AGENTS»
5   //arbitrary_agents
6   for ( Object obj : context ) {
7    if(obj instanceof «diseaseModel.agent.name») {
8     «diseaseModel.agent.name» agent = («diseaseModel.agent.name») obj;
9     agent.setGrid(this.grid);
10    «LET this.concerns.select(a | a.outputDatasets.size > 0) AS concerns-»
11     «FOREACH concerns AS concernOutPut»
12     «LET (mm::Concern) concernOutPut AS concern»
13      «FOREACH concern.outputDatasets AS outputDataSet»
14       «FOREACH outputDataSet.outputData AS outPutData»
15        «IF outPutData.entity.name.toString().toLowerCase().trim() ==
16             diseaseModel.agent.name.toString().toLowerCase().trim()»
17         «IF outPutData.metaType == Aggregation»
18         «LET (mm::Aggregation) outPutData AS aggregation»
19          «IF aggregation.aggregationType.toString().toUpperCase() == "COUNT"»
20           «IF aggregation.name == "susceptiveis"»
21              agent.updateCompartment();
22           «ENDIF»
23          «ENDIF»
24         «ENDLET»
25         «ENDIF»
26        «ENDIF»
27       «ENDFOREACH»
28      «ENDFOREACH»
29     «ENDLET»
30     «ENDFOREACH»
31    «ENDLET»
32    if(agent.getDisease«diseaseModel.id»().getCompartmental() ==
33    '«((DiseaseCompartment)sourceState.value).name»'){
34      candidateAgents.add(agent);
35    }
36   }
37  }
38  «ELSE»
39   //eligible
40   for ( Object obj : context ) {
41    if(obj instanceof «diseaseModel.agent.name») {
42     «diseaseModel.agent.name» agent = («diseaseModel.agent.name») obj;
43     agent.setGrid(this.grid);
44     if(agent.getDisease«diseaseModel.id»().getCompartmental() == '«((DiseaseCompartment)sourceState.value).name»'
45      && « EXpand sourceValue (diseaseModel.introduction.selectionCriteriaValue)   FOR diseaseModel.agent-»){
46      candidateAgents.add(agent);
47     }
48    }
49   }
50  «ENDIF»
51  if( candidateAgents.size() >= «EXpand sourceValue (diseaseModel.introduction.quantityTypeValue) for
52  diseaseModel.agent-»){
53   Random random = new Random();
54   for(int i = 0; i < «EXpand sourceValue (diseaseModel.introduction.quantityTypeValue) FOR
55   diseaseModel.agent-»; i++){
56    int sorteado = random.nextInt(candidateAgents.size());
57    selectedAgents.add(candidateAgents.get(sorteado));
58    candidateAgents.remove(sorteado);
59   }
60  }else{
61   selectedAgents = candidateAgents;
62  }
63 «ENDIF»
```

Fonte: Elaborado pela autora.

Java em que é terá um laço de repetição para percorrer todos os agentes da simulação. Se a probabilidade gerada de modo randômico for menor que a especificada na introdução da doença, o agente se torna um agente escolhido. Se for *eligible* além da probabilidade, o agente também deverá satisfazer o critério para então se tornar um agente escolhido (linhas 17 a 20).

Após os agentes serem criados, é verificado se possuem a doença. Caso a tenham, é chamado o método da Figura 41, assim como a cada *tick* da simulação através do método *step* da classe *Space*. Nele é verificado se a *periodicity* é *aperiodic* (linha 7) para então

Figura 40 – Código Xpand da introdução dos agentes infectados no modo probabilístico

```
1  «IF diseaseModel.introduction.quantityType == mm::IntroductedQuantityType::PROBABILISTIC»
2  //probabilistic
3  //selecionar uma quantidade de agentes
4  «IF diseaseModel.introduction.selectionCriteria == mm::IntroductionSelectionCriteria::ARBITRARY_AGENTS»
5  //arbitrary_agents
6   for(Object obj : context){
7    «diseaseModel.agent.name» agent = («diseaseModel.agent.name») obj;
8    if(Math.random() <= «EXpand sourceValue (diseaseModel.introduction.quantityTypeValue) FOR
9    diseaseModel.agent» && agent.getDisease«diseaseModel.id»().getCompartmental() ==
10         '«((DiseaseCompartment)sourceState.value).name»'){
11     selectedAgents.add(agent);
12   }
13 }«ELSE»
14 //eligible
15 for(Object obj : context){
16  «diseaseModel.agent.name» agent = («diseaseModel.agent.name») obj;
17  if(Math.random() <= «EXpand sourceValue (diseaseModel.introduction.quantityTypeValue) FOR
18  diseaseModel.agent» & agent.getDisease«diseaseModel.id»().getCompartmental() =
19  "«((DiseaseCompartment)sourceState.value).name»" & «EXpand sourceValue
20  (diseaseModel.introduction.selectionCriteriaValue) FOR diseaseModel.agent-»){
21     selectedAgents.add(agent);
22   }
23 }
24 «ENDIF»«ENDIF»
```

Fonte: Elaborado pela autora.

verificar se está no *tick* de introdução e então chamar o método que contém os códigos das Figuras 39 e 40. Se for *periodic* (linha 17) também é feito a verificação do *tick* com o especificado no diagrama. Esse último será chamado em todos os *ticks*.

Figura 41 – Código Xpand da introdução dos agentes de acordo com a periodicidade

```
1  private void DiseaseIntroduction(){
2  «LET (List[mm::Agent]) this.concerns.entities.select( a | mm::Agent.isInstance(a) &&
3  ((mm::Agent)a).capabilities.exists( c | mm::DiseaseModel.isInstance((c)))) AS agentsWithDiseaseModel»
4  «IF ! agentsWithDiseaseModel.isEmpty»
5    «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select( c |
6    mm::DiseaseModel.isInstance(c) ) AS diseaseModel»
7      «IF diseaseModel.introduction.periodicity == mm::IntroductionPeriodicity::APERIODIC»
8          //aperiodic
9          if(Disease«diseaseModel.agent.name»PeriodicityValue != 0 ){
10           if(RunEnvironment.getInstance().getCurrentSchedule().getTickCount() ==
11           Disease«diseaseModel.agent.name»PeriodicityValue){
12           //introdução de todos os agentes
13              diseaseIntroduction«diseaseModel.id»«diseaseModel.agent.name»();
14           }
15         }
16     «ELSE»
17           //periodic
18         if(RunEnvironment.getInstance().getCurrentSchedule().getTickCount() > 0 &&
19          ((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() %
20          Disease«diseaseModel.agent.name»PeriodicityValue) == 0)){
21           diseaseIntroduction«diseaseModel.id»«diseaseModel.agent.name»();
22         }
23     «ENDIF»
24   «ENDFOREACH»
25 «ENDIF»
26 «ENDLET»
27 }
```

Fonte: Elaborado pela autora.

3.2.4 Transmissão

A transmissão é sempre por probabilidade. No método da doença do agente, na linha 18, é chamado o método *infect()* (Figura 42) da classe da doença, sendo passado o agente infectado e o agente que pode ser infectado. O código Xpand que gera o método infect()

é apresentado na Figura 40. Dentro do método *infect()*, é chamado um outro método denominado *getTransitionValueTrigger()* onde é passado o compartimento suscetível, o agente infectado e o agente suscetível.

Figura 42 – Código Xpand do método infect()

```
1 «DEFINE infect (mm::DiseaseModel diseaseModel) FOR mm::AgentBasedSimulation »
2 public boolean infect(Object agent, Object suscetível) {
3         return getTransitionValueTrigger('S', agent, suscetível);
4 }
5 «ENDDEFINE»
```

Fonte: Elaborado pela autora.

O método *getTransitionValueTrigger()* é responsável por realizar todas as transições entre os compartimentos, incluindo a verificação da mortalidade. Um exemplo de código Java gerado a partir do gerador de código é mostrado na Figura 43. O argumento *compartimento* é utilizado para verificar qual o tipo de transição será feita. O argumento *suscetível* será o agente que poderá sofrer a progressão de compartimento, ou será infectado ou passará de um compartimento para o outro, como do "I" para o "R". O argumento *infected* será utilizado na hora da verificação da transmissão da doença, pois se houver mais de um tipo de agente que poderá infectar o agente suscetível, terá mais blocos de código, como será explicado a seguir. O código Xpand que gerá esse código será detalhado na próxima seção.

Figura 43 – Código Java da transition entre compartimentos

```
1  public boolean getTransitionValueTrigger(char compartimento, Object infected, Object suscetível) {
2    if (suscetível instanceof Pessoa) {
3      if ('S' == compartimento && (infected instanceof Pessoa)) {
4        if (Math.random() <= 0.4 && !die) {
5          return true;
6        }
7      }
8      if ('I' == compartimento) {
9        if (duration_current_I >= 10 && !die) {
10         if (Math.random() <= mortalityRatePessoa) {
11           die = true;
12           return true;
13         }
14         nextCompartimental();
15         return false;
16       } else {
17         duration_current_I++;
18       }
19     }
20   }
21   return false;
22 }
```

Fonte: Elaborado pela autora.

Como esse método faz todas as transições, nesse exemplo as linhas 2 a 7 serão detalhadas, pois é a parte de transmissão da doença. Para um agente infectar o outro, será passado como parâmetro um 'S'. Se houver mais de um agente, o bloco de código da linha 2 a 7 irá se repetir, mudando apenas a comparação da instância do objeto *infected*.

No exemplo da Figura 43, é iniciado com a verificação da instância do agente suscetível (linha 2) e se o compartimento desse agente se encontra no "S" e se o agente

infectante é do tipo Pessoa (linha 3). Ao satisfazer as condições, então um número aleatório será gerado e se ele for menor ou igual à probabilidade de transmissão, retornará um *true*, indicando que o agente suscetível será infectado. No mesmo método é realizada a progressão da doença.

No mesmo exemplo do mencionado no paragráfo anterior, temos a progressão do compartimento "I" (linha 8). Após entrar dentro do *IF* correspondente, é verificado quanto tempo o agente se encontra naquele compartimento (linha 9). Caso o tempo de permanência atinja a duração especificada na doença, que nesse caso é um tempo igual a 10, será feita a progressão da doença. Caso contrário, a sua duração será incrementada. Esse processo será explicado com maiores detalhes na próxima seção. A mortalidade da doença foi configurada como "ao sair do compartimento", deste modo, primeiro é verificado a condição da mortalidade (linha 10). Se a condição não for satisfeita, o agente chamará o método *nextCompartimental()*. Mais detalhes sobre a mortalidade serão apresentados na seção 3.2.6.

3.2.5 Progressão

A progressão da doença é realizada no mesmo método citado anteriormente (Figura 43). No código da doença na classe do agente (Figura 32) na linha 28 é chamado o método *transitionBetweenStates* da classe da doença (Figura 44), sendo passado como parâmetro o próprio agente. Esse método será chamado na classe do agente sempre que puder ocorrer uma transição. Nele será feito um *foreach* para percorrer todos os agentes que possuem a doença, para que o bloco de código das linhas 11 a 13 se repitam. Nele será verificado o tipo do agente passado como parâmetro (linha 11) e chamar outro método, o *getTransitionValueTrigger()* (linha 12), que passará como parâmetro o compartimento atual da doença e o próprio agente. o Método *getTransitionValueTrigger()* faz a progressão de qualquer compartimento além de realizar a transmissão da doença, por isso ele recebe dois agentes como argumento, porém como não é uma transição de transmissão, o outro agente será passada como *null*.

A Figura 45 mostra o código Xpand do método *getTransitionValueTrigger()*, que é responsável por gerar o código Java da Figura 43. Nas linhas 1 e 2 são geradas as condições de comparação dos compartimentos que a doença pode ter. Nas linhas 3, 22 e 36 é verificado qual o tipo de progressão que foi específicado no no modelo da simulação para cada compartimento para gerar o código da condição correspondente, através da verificação do *kind* da *trigger* da *transition*. Das linhas 7 a 21 é gerado o código para o tipo determinístico, da 24 a 35 a condicional e por último da 38 a 51 o probabilístico.

Figura 44 – Código Xpand da do método *transitionBetweenStates*

```
1  public boolean transitionBetweenStates(Object agent){
2  «LET this.concerns.entities.select(a | a.metaType == Agent) AS agents-»
3  «FOREACH agents AS entity»
4    «LET (mm::Agent)entity AS agent»
5    «LET (List[mm::Agent]) this.concerns.entities.select( a | mm::Agent.isInstance(a) &&
6      ((mm::Agent)a).capabilities.exists( c | mm::DiseaseModel.isInstance((c)) )) AS agentsWithDiseaseModel»
7    «IF !agentsWithDiseaseModel.isEmpty»
8      «FOREACH (List[mm::DiseaseModel]) agentsWithDiseaseModel.capabilities.select( c |
9        mm::DiseaseModel.isInstance(c) ) AS diseaseModelAgent»
10       «IF diseaseModelAgent.id == diseaseModel.id && agent.name == diseaseModelAgent.agent.name»
11         if(agent instanceof «diseaseModelAgent.agent.name»){
12           getTransitionValueTrigger(this.compartmental_«diseaseModelAgent.agent.name», null, agent);
13         }
14       «ENDIF»
15     «ENDFOREACH»
16   «ENDIF»
17   «ENDLET»
18   «ENDLET»
19 «ENDFOREACH»
20 «ENDLET»
21   return this.die;
22 }
```

Fonte: Elaborado pela autora.

Figura 45 – Código Xpand da progressão entre os compartimentos

```
1  if('«state.name»' == compartimento «IF state.name == 'S'» && (infected instanceof
2  «interactionExp.infectiousAgentDiseaseModel.agent.name»)«ENDIF»){
3  «IF ((StandardTrigger)transition.trigger).kind == TriggerKind::TIME »
4    «LET (ExpressionSource) ((StandardTrigger)transition.trigger).value AS triggerExpSrc»
5    «LET (StateMachineStateTimeExpression) triggerExpSrc.expression AS triggerExp»
6    «EXpand mortalityType ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
7    FOR this»
8     if(duration_current_«state.name» >= «EXpand sourceValue(((StandardTrigger)transition.trigger).value) FOR
9     diseaseModel.agent» && !die){
10    «EXpand mortalityTypeLeaving ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
11    FOR this»
12      nextCompartimental();
13      return false;
14    }
15   «IF state.name != 'S'»else{
16    duration_current_«state.name»++;
17    }
18   «ENDIF»
19 «ENDLET»
20 «ENDLET»
21 «ENDIF»
22 «IF ((StandardTrigger)transition.trigger).kind == TriggerKind::CONDITION »
23 //CONDICIONAL
24 «EXpand mortalityType ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
25 FOR this»
26 if(«EXpand sourceValue(((StandardTrigger)transition.trigger).value) FOR diseaseModel.agent» «IF
27 transition.guard != null-» &&
28  «EXpand guardExpressionValue(transition) FOR diseaseModel-»«ENDIF-» && !die){
29  «EXpand mortalityTypeLeaving ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
30  FOR this»
31   nextCompartimental();
32   return false;
33  }«IF state.name != 'S'»else{
34   duration_current_«state.name»++;
35   }«ENDIF»«ENDIF»
36 «IF ((StandardTrigger)transition.trigger).kind == TriggerKind::PROBABILITY »
37 //PROBABILITY
38 «EXpand mortalityType ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
39 FOR this»
40 if(Math.random() <= «EXpand sourceValue(((StandardTrigger)transition.trigger).value) FOR diseaseModel.agent»
41 && !die){
42  «EXpand mortalityTypeLeaving ((DiseaseModel)diseaseModelAgent, diseaseModelAgent.agent, state.name.toString())
43  FOR this»
44  «IF state.name != 'S'»
45   nextCompartimental();
46   return false;
47 «ELSE»return true;«ENDIF»
48  }«IF state.name != 'S'»else{
49   duration_current_«state.name»++;
50  }«ENDIF»«ENDIF»
51 }
```

Fonte: Elaborado pela autora.

No início de cada tipo de condição é chamado o *template mortalityType* (linhas 6, 24 e 38) que gerará a condição da mortalidade, caso ela for do tipo “quando a condição

ocorrer", "a todo momento" ou "em um momento específico após infectado". Caso for o tipo "ao sair do compartimento", será chamado o *template mortalityTypeLeaving*, que ficará dentro do código que satisfaça a condição de progressão do compartimento (linhas 10, 29 e 42).

Na duração do tipo determinístico é verificado se a duração do atual compartimento chegou no tempo de progredir para o próximo de acordo com as configurações do diagrama (linhas 8e 9). No tipo condicional é verificado se a condição do diagrama foi satisfeita (linhas 26 a 28). Por fim, no tipo probabilístico, se foi gerado um número menor que a probabilidade do diagrama (linhas 40 e 41). Os valores são retornados através da chamada do *template sourceValue* que verifica o *value* presente dentro do *trigger* da *transition*.

3.2.6 Mortalidade

Existem quatro tipos de mortalidade, para diferenciar a mortalidade do tipo "ao sair do compartimento' utilizou-se o nome do *state*, que é *whenleavingcompartmentI* ou *whenleavingcompartmentE*, enquanto que os outros possuem o nome do seu compartimento no nome do seu *state*. O trecho de código do *template mortalityTypeLeaving* pode ser visto na Figura 46. Nele é gerado o código Java que fará a condição da mortalidade do tipo "ao sair do compartimento'.

Figura 46 – Código Xpand da mortalidade "Ao sair do compartimento"

```
1  «FOREACH diseaseModelAgent.states AS state»
2   «FOREACH state.outgoing.select( t | StandardTrigger.isInstance(t.trigger) ) AS transition»
3    «IF transition.target.name == "dead" && (stateAgent == "I" || stateAgent == "E")»
4     «IF state.name == "whenleavingcompartmentI" || state.name == "whenleavingcompartmentE"»
5      if(Math.random() <= mortalityRate«diseaseModelAgent.agent.name»I){
6        die = true;
7        return true;
8      }
9  «ENDIF»«ENDIF»«ENDFOREACH»«ENDFOREACH»
```

Fonte: Elaborado pela autora.

O código é iniciado fazendo dois *foreach*, para percorrer todos os *states* da doença e das *transition* de cada *state* para então verificar o alvo da transição (linha 3). Se for uma mortalidade, terá o nome do alvo como *dead*, e como estamos falando da mortalidade "ao sair do compartimento", também é feita a verificação para saber se o *stateAgent*, que foi passado como parâmetro é o "I" ou o "E". Na linha 4 é feita uma verificação do nome do *state*, para então gerar o trecho de código das linhas 5 a 8.

Como mencionado anteriomente, os outros tipos de mortalidade terão como nome do *state* o seu compartimento e serão gerados em outro trecho de código. Devido a isso, foi criado um outro *template* chamado *mortalityType* para as condições serem geradas. O trecho de código desse *template* pode ser visto na Figura 47.

Do mesmo modo que o código da Figura 46, também irá percorrer os *states* da doença e suas *transition*. Na linha 3 é verificado o nome do alvo da transição, que deverá ser *dead*. A mortalidade do tipo "em um momento específico após a infecção" possui um *TimeExpression* na *guard* da transição. Desta forma foi utilizada essa condição para diferenciá-la das demais (linha 4), que gerará o trecho das linhas 5 a 11, que verifica se a duração no compartimento e a probabilidade gerada chegaram na especificada pelo modelo da simulação. Na linha 12 é feito uma comparação com o valor da *guard*. Se for igual a *null*, então a duração é do tipo "a todo momento", caso contrário será "quando a condição ocorrer". Nesse último, o código Java que será gerado primeiro fará a verificação da condição presente dentro da *guard* e depois verificará a probabilidade de morte.

Figura 47 – Parte do código Xpand dos três tipos de mortalidade

```
1  «FOREACH diseaseModelAgent.states AS state»
2   «FOREACH state.outgoing.select( t | StandardTrigger.isInstance(t.trigger) ) AS transition»
3    «IF state.name == stateAgent && transition.target.name == "dead"»
4     «IF StateMachineStateTimeExpression.isInstance(transition.guard)»
5     if(this.duration_current_«state.name»_«diseaseModelAgent.agent.name»  ==
6     timeUnity«diseaseModelAgent.agent.name»){
7      if(Math.random() < mortalityRate«diseaseModelAgent.agent.name»«state.name»){
8        die = true;
9        return true;
10     }
11    }«ELSE»
12    «IF transition.guard.value == null»
13    if(Math.random() < mortalityRate«diseaseModelAgent.agent.name»«state.name»){
14      die = true;
15      return true;
16    }«ELSE»
17    if(«transition.guard.value»){
18     if(Math.random() < mortalityRate«diseaseModelAgent.agent.name»«state.name»){
19      die = true;
20      return true;
21     }
22    }
23 «ENDIF»«ENDIF»«ENDIF»«ENDFOREACH»«ENDFOREACH»
```

Fonte: Elaborado pela autora.

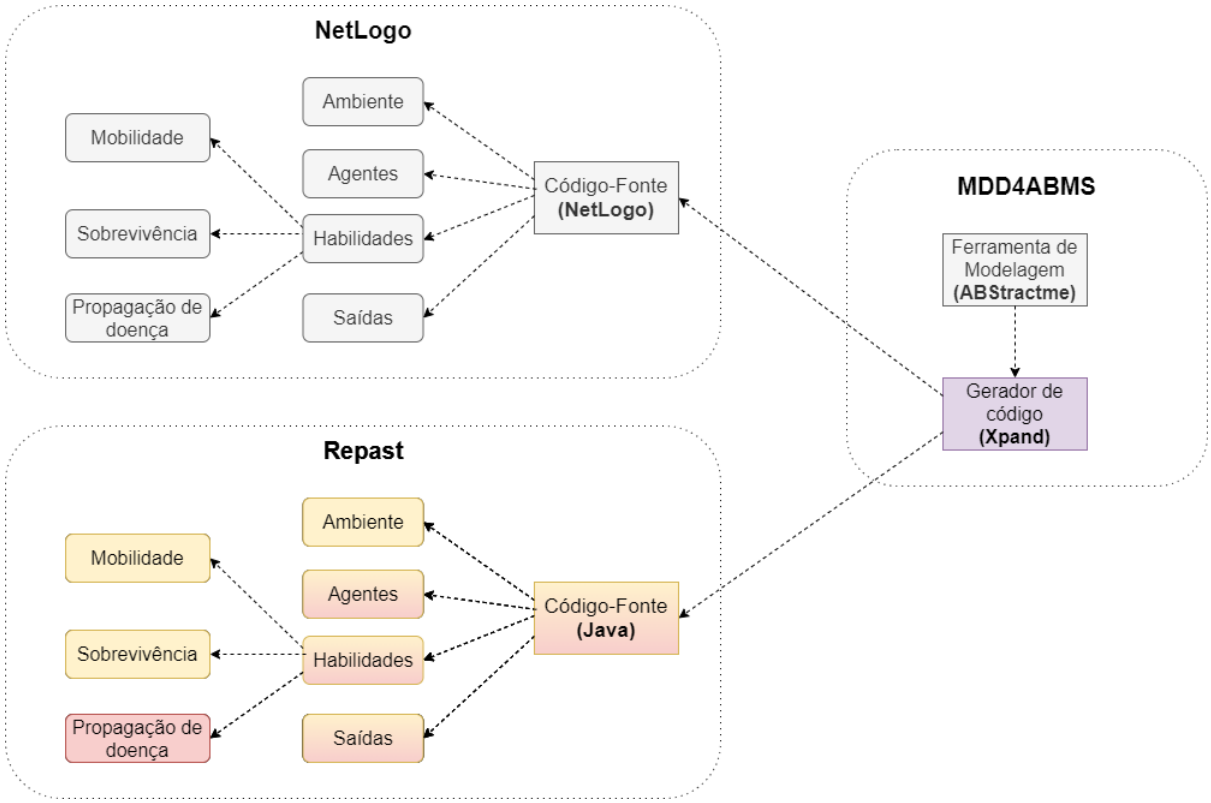
3.3 CONSIDERAÇÕES FINAIS SOBRE A GERAÇÃO DE CÓDIGO

A abordagem MDD4ABMS, quando desenvolvida por Santos (2019), já gerava código para a plataforma NetLogo. Tenfen (2019) desenvolveu o gerador de código para gerar código-fonte Java para a plataforma de simulação Repast dos seguintes aspectos da simulação: ambiente *grid* de duas dimensões; agentes; saídas da simulação; e as habilidade de mobilidade e de sobrevivência. O presente trabalho desenvolveu o gerador de código-fonte para gerar código Java para a plataforma Repast referente a habilidade de propagação de doença e ao gráfico de saída para a contagem de agentes em cada compartimento.

Na Figura 48 temos uma visão geral da abordagem MDD4ABMS após o desenvolvimento deste trabalho. Essa abordagem utiliza a ferramenta de modelagem ABStractme e com o gerador de código Xpand gerá o código-fonte para a plataforma de simulação. Os

itens do diagrama que estão em cinza claro são os itens que já estavam prontos na abordagem MDD4ABMS. Os itens em amarelo são os desenvolvidos por Tenfen (2019). Por fim os itens em vermelho são os desenvolvidos totalmente nesse trabalho. Os itens em degradê amarelo/vermelho são aqueles itens desenvolvidos por Tenfen (2019) que foram modificados nesse trabalho para acrescentar a habilidade de propagação de doença. O gerador de código está em roxo para indicar que é nele que todo o trabalho foi desenvolvido.

Figura 48 – Visão Geral do gerador de código



Fonte: Elaborado pela autora.

4 AVALIAÇÃO DA PORTABILIDADE

Para avaliar a portabilidade do modelo de propagação de doenças, foram modeladas duas simulações. Cada simulação utiliza um determinado modelo compartimental. As simulações foram modeladas na ferramenta ABStracme, e dois códigos fontes foram gerados, sendo um para NetLogo, e outro para Repast. Estes códigos foram executados, e os resultados obtidos foram comparados para verificar se as simulações produzem resultados equivalentes.

4.1 ESPECIFICAÇÃO DAS SIMULAÇÕES

Foram escolhidos dois modelos compartimentais para demonstrar o funcionamento da portabilidade proposta nesse trabalho, o modelo SIR e o SEIR. Na Tabela 3 podemos comparar os compartimentos e os agentes de cada modelo. O modelo SEIR possui o compartimento Exposto a mais que o modelo SIR e ambos possuem os agentes Humano e Pet com quantidade de 200 e 100 agentes, respectivamente.

Tabela 3 – Compartimentos das simulações com os modelos SIR e SEIR

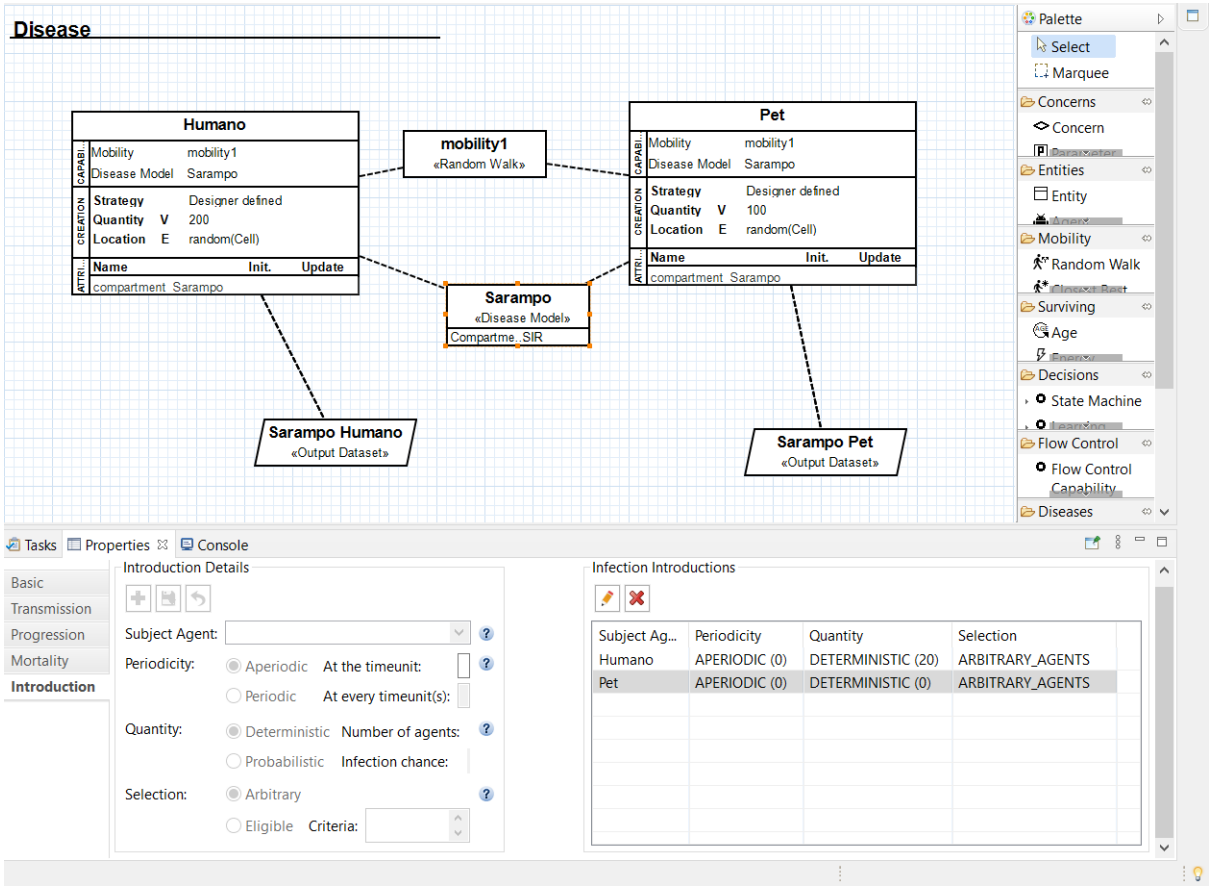
Modelo Compartimental	Compartimentos	Tipos de agentes	Quantidade de agentes
SIR	Suscetível	Humano	200
	Infectado	Pet	100
	Recuperado		
SEIR	Suscetível	Humano	200
	Exposto	Pet	100
	Infectado		
	Recuperado		

Fonte: Elaborado pela autora.

A Figura 49 mostra o diagrama onde a simulação foi modelada. Foram configurados dois agentes, Humano e Pet, em que ambos possuem a habilidade de se mover e a de propagação da doença chamada Sarampo. Ambos os agentes possuem gráficos de saída. Abaixo do diagrama, temos a aba *properties* onde são realizadas todas as configurações da doença, como a progressão, através das suas subabas. No exemplo mostrado, está aberta a subaba *introduction*, onde é realizada a configuração da introdução da doença.

A especificação utilizada para a introdução da doença é mostrada na Tabela 4. Como pode-se verificar, em ambas as simulações a introdução se dá no início da simulação, no *tick* 0 com seleção do número de agentes de maneira determinística de 20 e 10 agentes

Figura 49 – Diagrama da modelagem da simulação com os agentes Humano e Pet e a doença Sarampo na ferramenta ABStractme



Fonte: Elaborado pela autora.

Humanos para as simulações SIR e SEIR, respectivamente. Em ambas, os agentes que serão infectados no início da simulação são escolhidos de modo aleatório.

Tabela 4 – Introdução da doença nos modelos SIR e SEIR

Modelo Compartimental	Tipos de agentes	Periodicidade	Quantidade de agentes	Seleção	Modo
SIR	Humano Pet	*Aperiodic* (0) -	20 -	*deterministic* -	*arbitrary*
SEIR	Humano Pet	*Aperiodic* (0) -	10 -	*deterministic* -	*arbitrary*

Fonte: Elaborado pela autora.

Na Tabela 5, são detalhadas as transmissões especificadas entre os tipos de agentes. Na simulação que usa o modelo SIR, por exemplo, a probabilidade de transmissão de um pet infectado para um humano é de 0.3 por contato. É possível observar que em ambos modelos, as transmissões por contato são entre agentes diferentes, enquanto que

por próximidade de 2 unidades distâncias é entre agentes da mesma classe. Os tipos de transmissão escolhidos foi uma decisão arbitrária para fim de teste.

Tabela 5 – Transmissão da doença nos modelos SIR e SEIR

Modelo Compartimental	Interação	Tipo	Distância	Probabilidade
SIR	Pet → *Humano*	contato	0	0.3
	Pet → *Pet*	proximidade	2	0.6
	Humano → *Humano*	proximidade	2	0.6
	Humano → *Pet*	contato	0	0.5
SEIR	Pet → *Humano*	contato	0	0.3
	Pet → *Pet*	proximidade	2	0.4
	Humano → *Humano*	proximidade	2	0.6
	Humano → *Pet*	contato	0	0.5

Fonte: Elaborado pela autora.

A progressão da doença em todos foi determinística, com duração de 10 *ticks* nos humanos para o compartimento I em ambos modelos. Na Tabela 6 são mostradas todas as progressões dos dois modelos compartimentais.

Tabela 6 – Progressão da doença nos modelos SIR e SEIR

Modelo Compartimental	Tipos de agentes	Compartimento	Tipo	Duração
SIR	Humano	I	*deterministic*	10
	Pet	I	*deterministic*	15
SEIR	Humano	I	*deterministic*	10
		E	*deterministic*	3
	Pet	I	*deterministic*	10
		E	*deterministic*	3

Fonte: Elaborado pela autora.

A mortalidade pode ser vista na Tabela 7. Ambos modelos compartimentais foram configurados iguais.

Tabela 7 – Mortalidade da doença nos modelos SIR e SEIR

Modelo Compartimental	Tipos de agentes	Compartimento	Probabilidade	Tipo
SIR	Humano	I	0.2	*when leaving compartiments*
SEIR	Humano	I	0.2	*when leaving compartiments*

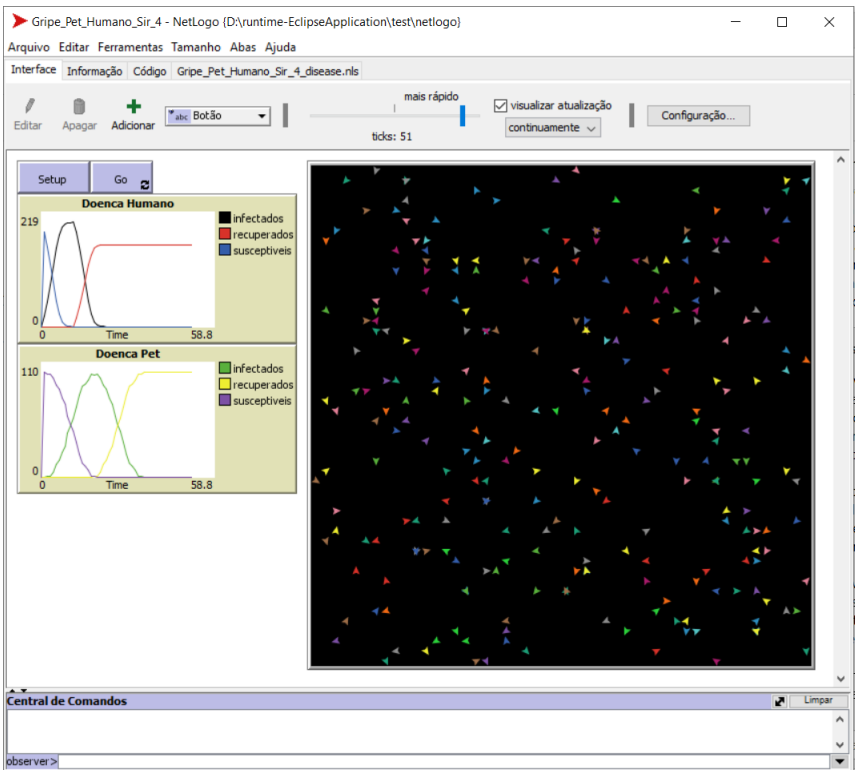
Fonte: Elaborado pela autora.

4.2 RESULTADOS OBTIDOS

Após todas as especificações destas configurações da doença através da ferramenta ABStractme estarem prontas, foram gerados os códigos para as plataformas Repast e NetLogo. Foi especificada como saída das simulações a contagem de agentes em cada compartimento para ambos agentes, Pet e Humano.

Nas Figuras 50 e 51 temos as simulações do modelo compartimental SIR nas plataformas NetLogo e Repast, respectivamente. Em relação a plataforma NetLogo, temos os gráficos de saída ao lado esquerdo, a simulação com os agentes no centro e no painel acima da simulação temos os *ticks* da simulação, que se encontra no 51. Pode-se controlar a velocidade no componente de arrastar, acima do número de *ticks*.

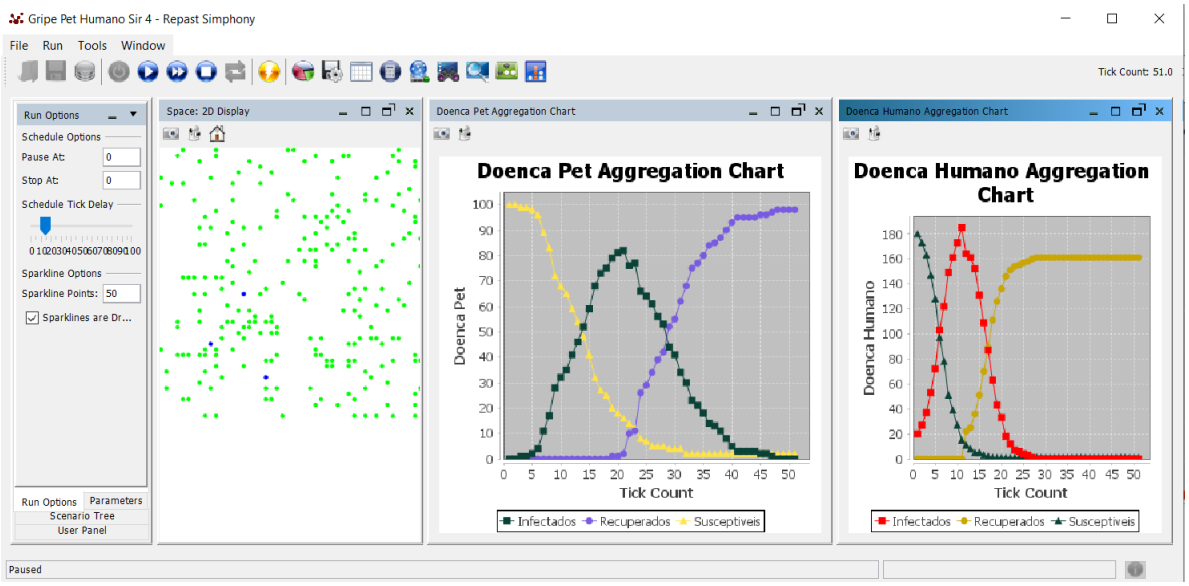
Figura 50 – Simulação do modelo compartimental SIR na plataforma NetLogo



Fonte: Elaborado pela autora.

Em relação à simulação na plataforma Repast, temos o primeiro bloco que controla a velocidade e podemos controlar outros parâmetros da simulação também. Ao lado temos o ambiente de simulação com os agentes, em que os verde representam aqueles agentes que estão recuperados e os de azul os suscetíveis. Os dois últimos blocos são os gráficos de saída. Acima do gráfico do Humano, temos a contagem dos *ticks* da simulação, que se encontra no 51.

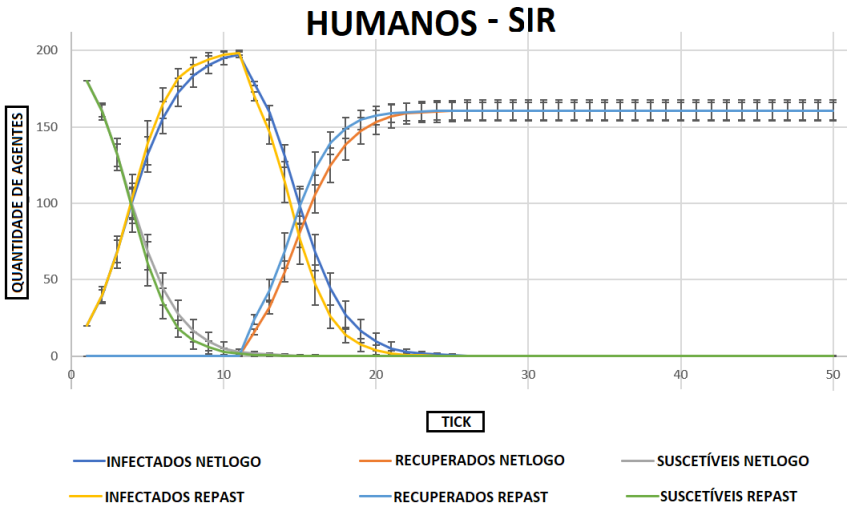
Figura 51 – Simulação do modelo compartimental SIR na plataforma Repast



Fonte: Elaborado pela autora.

Para comparar os resultados produzidos pelos códigos NetLogo e Repast, cada simulação foi executada 20 vezes, e os resultados foram coletados e tabulados no Excel. Em seguida, foram calculados a média e o desvio padrão de cada compartimento. Na Figura 52 temos o gráficos do agente Humano do modelo compartimental SIR. Nele são apresentados as linhas com as quantidades de agentes em cada compartimento em ambas plataformas com seu desvio-padrão.

Figura 52 – Gráfico das médias e desvio-padrão do modelo SIR do Agente Humano

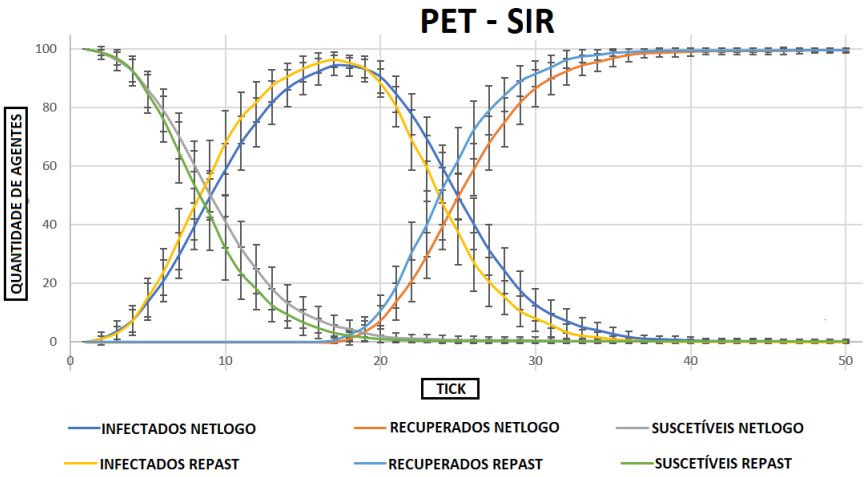


Fonte: Elaborado pela autora.

Na Figura 53 temos o gráfico do agente Pet do modelo compartimental SIR. Em ambos os gráficos é possível observar que as linhas das médias tiveram o mesmo padrão e ficaram dentro do desvio-padrão uma da outra. Como a progressão da doença passa

de suscetível para infectado e de infectado para recuperados, o comportamento da linha de suscetíveis é diminuir e não voltar a crescer. O contrário acontece para a linha dos recuperados.

Figura 53 – Gráfico das médias e desvio-padrão do modelo SIR do Agente Pet



Fonte: Elaborado pela autora.

No gráfico dos humanos, é iniciado com 20 agentes infectados enquanto que no dos PET com 0. Na linha dos infectados podemos observar que ela sobe e após 10 *ticks* no gráfico dos humanos e 15 no dos pets ela começa a cair e a linha dos recuperados começa a aumentar devido aos primeiros agentes infectados chegarem no tempo de progressão, configurado no diagrama. Ambos agentes chegam no pico máximo de infecção para então começar a decrescer.

É possível observar que o número de agentes recuperados do tipo Humano não chega ao número máximo de agentes, ou seja, 200 agentes, pois como existe a mortalidade, alguns acabam morrendo. Já os agentes do tipo Pet como não existe a mortalidade, todos acabam se recuperando.

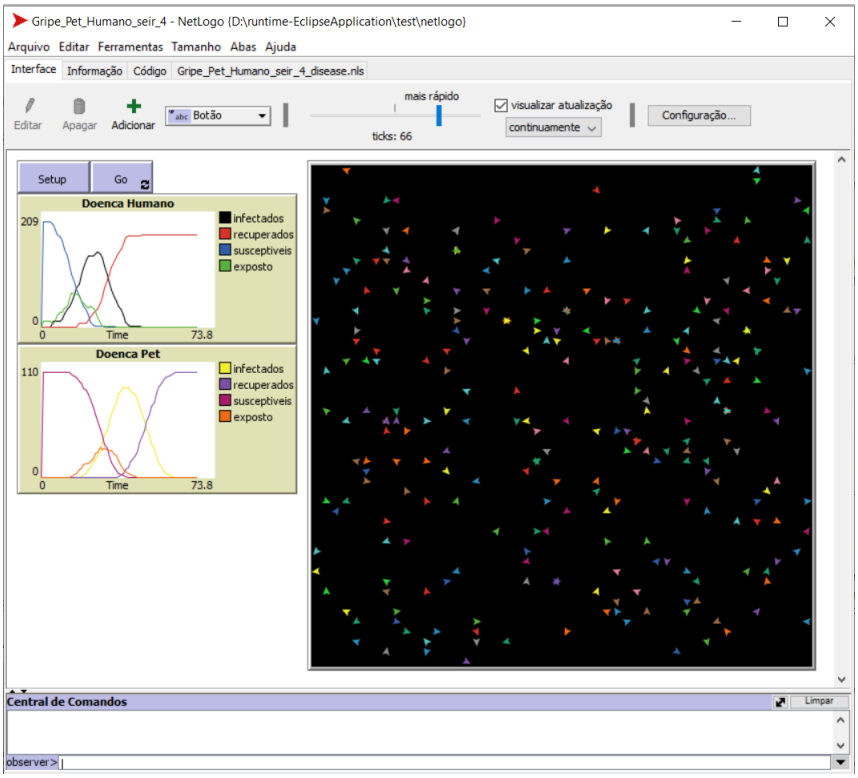
Nas Figuras 54 e 55 são apresentados os gráficos de saída modelo compartimental SEIR nas plataformas NetLogo e Repast, respectivamente.

O gráfico gerado pelo excel do agente humano pode ser visto na Figura 56 e do pet na Figura 57, onde é possível observar que eles possuem uma linha a mais do que o modelo anterior, devido ao compartimento adicional "exposto".

Em ambos os gráficos podemos observar que o comportamento da curva de cada compartimento segue o mesmo padrão. Apesar de em alguns momentos as linhas ficarem mais distante, os desvios-padrões de cada plataforma continuam no limite uma da outra.

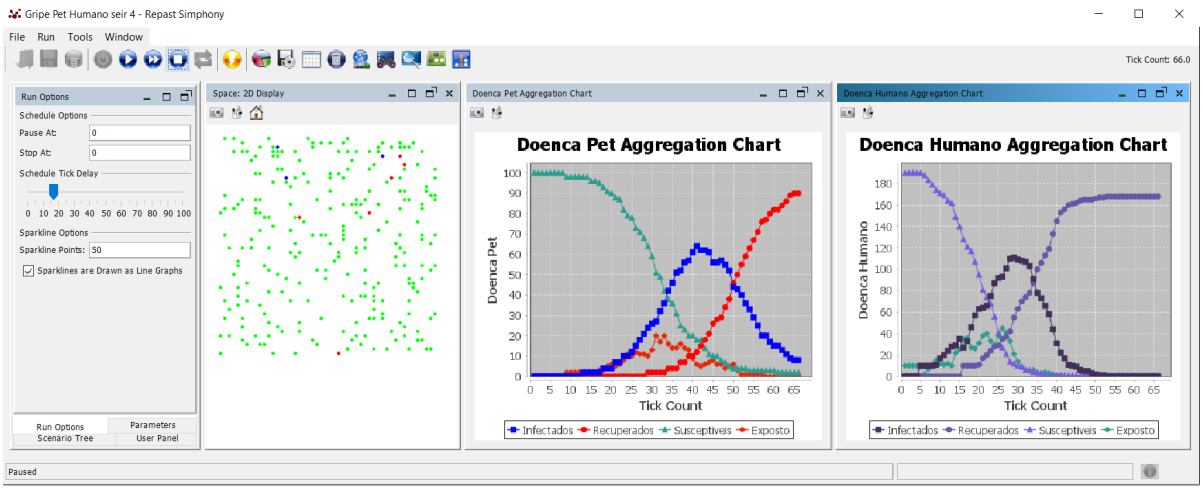
Conforme demostrado nesse estudo de caso das simulações dos modelos compartimentais SIR e SEIR com os agentes Humanos e Pet, os gráficos gerados mostraram

Figura 54 – Simulação do modelo compartimental SEIR na plataforma NetLogo



Fonte: Elaborado pela autora.

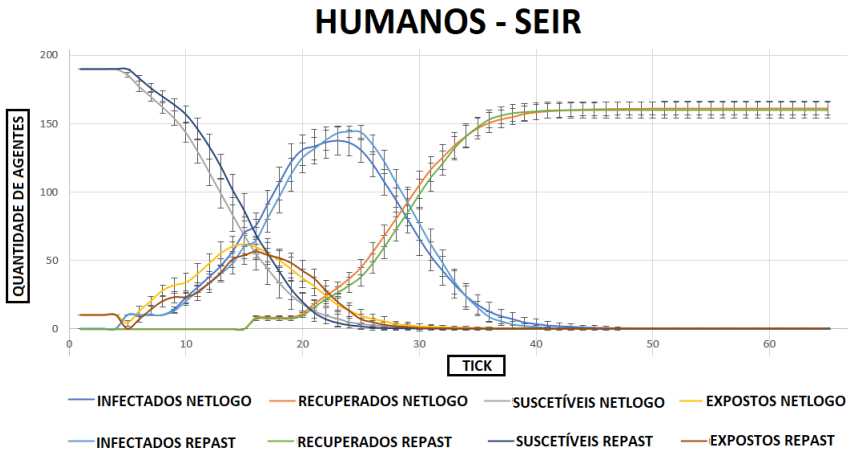
Figura 55 – Simulação do modelo compartimental SEIR na plataforma Repast



Fonte: Elaborado pela autora.

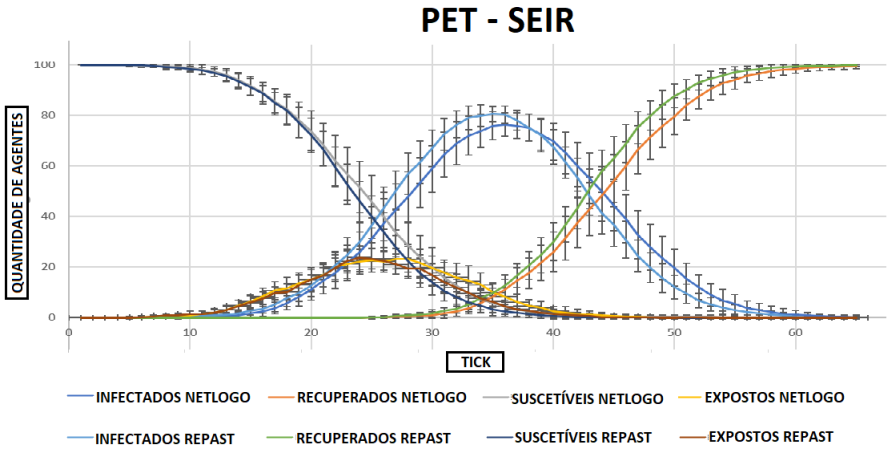
resultados semelhantes com os padrões das linhas de cada compartimento em ambos modelos sendo similares e os desvio-padrão de cada um ficou dentro do limite um do outro, demonstrando que é possível realizar a portabilidade da habilidade de propagação de doenças para as plataformas de simulações NetLogo e Repast a partir de um mesmo modelo.

Figura 56 – Gráfico das médias e desvio-padrão do modelo SEIR do Agente Humano



Fonte: Elaborado pela autora.

Figura 57 – Gráfico das médias e desvio-padrão do modelo SEIR do Agente Pet



Fonte: Elaborado pela autora.

5 CONCLUSÕES

Esse trabalho trouxe a portabilidade do modelo de propagação de doenças criados a partir da abordagem MDD4ABMS. Essa portabilidade permitiu gerar códigos para a plataforma de simulação Repast, auxiliando os projetistas que trabalham com essa plataforma no processo de desenvolvimento de simulações com agentes.

Para realizar a portabilidade para a plataforma de simulação de agentes Repast, foi adotada a abordagem MDD4ABMS, utilizando as ferramentas da mesma, como um plugin do Eclipse IDE. Foi aproveitada a lógica de geração do código fonte para a plataforma NetLogo por meio da linguagem Xpand porém com regras definidas para a geração de código Java para a plataforma Repast.

Para comprovar a corretude da portabilidade da propagação de doenças, foram analisados os gráficos de saída de dois modelos compartimentais nas plataformas NetLogo e Repast. Observou-se nessa análise que o padrão das linhas de cada compartimento em ambas plataformas foram similares e ficaram dentro do desvio-padrão uma da outra. Esse resultado evidência que é possível realizar a portabilidade da habilidade de propagação de doenças para as plataformas de simulações NetLogo e Repast a partir de um mesmo modelo.

O diferencial proposto nesse trabalho é a extensão do gerador de código para a plataforma Repast do modelo de propagação de doenças. No trabalho de Tenfen (2019) foi desenvolvida a portabilidade para a plataforma de simulação Repast de ambientes simulados do tipo *grid* de duas dimensões e algumas outras habilidades, porém não foi desenvolvido o modelo proposto nesse trabalho que é o modelo de propagação de doenças. A ferramenta EasyABMS de Garro e Russo (2010) também utiliza o Eclipse IDE e gera código fonte a partir de modelos para a plataforma Repast e sua modelagem é baseada na UML. Diferente na abordagem MDD4ABMS utilizada nesse trabalho, a EasyABMS não possui um nível elevado de abstração, ficando restrita à plataforma Repast.

A análise da portabilidade da geração da habilidade de propagação de doenças foi feita apenas com modelos compartimentais simples, como o SIR e SEIR. Para coletar evidências mais robustas da portabilidade da habilidade de propagação da doença seria necessário realizar novos estudos com os modelos compartimentais PSEIR e a Custom, por exemplo, ficando como sugestões de trabalho futuro. Outra sugestão de trabalho futuro é a progressão do tipo de duração *custom*.

REFERÊNCIAS

ATKINSON, C.; KÜHNE, T. Model-driven development: A metamodeling foundation. **IEEE Software**, IEEE Computer Society, 2003.

BARROS, J. P. S. de; BALDAM, R. de L.; JUNIOR, T. de P. C.; LEAL, E. de A. S.; SOUZA, M. A. V. F. de. Simulação baseada em agentes. In: **Enegep**. [S.l.: s.n.], 2011.

BUARQUE, A. Desenvolvimento de software dirigido a modelos. In: **Qualidade, Processos e Gestão de Software**. Pernambuco: UFPE, 2012.

COLLIER, N.; HOWE, T.; NORTH, M. Onward and upward: The transition to repast 2.0. In: **Proceedings of the First Annual North American Association for Computational Social and Organizational Science Conference**. Pittsburgh, USA: [s.n.], 2004.

COLLIER, N.; NORTH, M. **Repast Java Getting Started**. 2016. Disponível em: <https://repast.github.io/docs/RepastJavaGettingStarted.pdf>. Acesso em: 13 mar. 2020.

FRIESE, P. **Getting Started With Code Generation With Xpand**. 2010. Disponível em: <https://dzone.com/articles/getting-started-code>. Acesso em: 22 mar. 2020.

GALLARDO, M. del M.; MARTÍNEZ, J.; MERINO, P.; RODRÍGUEZ, G. Integration of reliability and performance analyses for active network services. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 133, p. 217–236, 2005.

GARRO, A.; RUSSO, W. Exploiting the easyabms methodology in the logistics domain. In: **Proceedings of the Second Multi-Agent Logics, Languages, and Organisations Federated Workshops**. [S.l.: s.n.], 2008.

GARRO, A.; RUSSO, W. easyabms: A domain-expert oriented methodology for agent-based modeling and simulation. **Simulation Modelling Practice and Theory**, Elsevier, v. 18, n. 10, p. 1453–1467, 2010.

ISERN, D.; MORENO, A. A systematic literature review of agents applied in healthcare. **Journal of medical systems**, Springer, v. 40, n. 2, p. 43, 2016.

KERMACK, W. O.; MCKENDRICK, A. G. Contributions to the mathematical theory of epidemics. ii. —the problem of endemicity. **The Royal Society**, Taylor & Francis, v. 138, n. 834, p. 55–83, 1932.

KLÜGL, F. A validation methodology for agent-based simulations. In: **Proceedings of the 2008 ACM symposium on Applied computing**. [S.l.: s.n.], 2008. p. 39–43.

KLÜGL, F.; BAZZAN, A. L. Agent-based modeling and simulation. **Ai Magazine**, v. 33, n. 3, p. 29–29, 2012.

LIMA, T. F. M.; FARIA, S. D.; FILHO, B. S. S.; CARNEIRO, T. C. S. Modelagem de sistemas baseada em agentes: alguns conceitos e ferramentas. In: **XIV Simpósio Brasileiro de Sensoriamento Remoto**. Natal: [s.n.], 2009. p. 5279–5286.

MACAL, C.; NORTH, M. Introductory tutorial: Agent-based modeling and simulation. In: IEEE. **Proceedings of the Winter Simulation Conference 2014**. [S.l.], 2014. p. 6–20.

MDD4ABMS. **MDD4ABMS-META**. 2017. Disponível em: <https://github.com/agentsimulations/mdd4abms-meta>. Acesso em: 17 mar. 2021.

MENDIX. **What is Model Driven Development (MDD)?** 2020. Disponível em: <https://www.mendix.com/model-driven-development/>. Acesso em: 29 abril. 2020.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005.

MOHAGHEGHI, P.; DEHLEN, V.; NEPLE, T. Definitions and approaches to model quality in model-based software development–a review of literature. **Information and software technology**, Elsevier, v. 51, n. 12, p. 1646–1669, 2009.

MOREIRA, D.; SANTOS, F.; BARBIERI, M.; NUNES, I.; BAZZAN, A. L. Abstractme: Modularized environment modeling in agent-based simulations. In: **Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems**. [S.l.: s.n.], 2017. p. 1802–1804.

NOGUEIRA, T. P.; SOUZA, C. T. de; CORTÉS, M. I. Geração de aplicações hipermídia para televisão digital baseada em desenvolvimento orientado a modelos. In: **XXXV Conferência Latinoamericana de Informática (CLEI)**. Ceará: Elsevier, 2008. p. 2–11.

NORTH, M. J.; COLLIER, N. T.; VOS, J. R. Experiences creating three implementations of the repast agent modeling toolkit. **ACM Transactions on Modeling and Computer Simulation (TOMACS)**, ACM, v. 16, n. 1, p. 1–25, 2006.

REPAST. **Recursive Porus Agent Simulation Toolkit**. 2008. Disponível em: <http://repast.sourceforge.net/repast\textunderscore3/index.html>. Acesso em: 13 mar. 2020.

REPAST. 2019. Disponível em: <https://repast.github.io/>. Acesso em: 13 mar. 2020.

RUSSEL, S.; NORVIG, P. **Inteligência Artificial**. [S.l.]: Elsevier, 2004.

SANTOS, F. **Model-driven Agent-based Simulation Development**. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul (UFRGS), 2019.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. Supporting the development of agent-based simulations: a dsl for environment modeling. In: IEEE. **2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)**. [S.l.], 2017. v. 1, p. 170–179.

SANTOS, F.; NUNES, I.; BAZZAN, A. L. C. Model-driven agent-based simulation development: A modeling language and empirical evaluation in the adaptive traffic signal control domain. **Simulation Modelling Pratice and Theory**, Elsevier, v. 83, p. 162–187, 2018.

SELIC, B. The pragmatics of model-driven development. **IEE Computer Society**, v. 20, n. 5, p. 19–25, 2003.

SEVERIEN, A. L. **Web Application Language Engine (WALE): Geração de código para aplicações J2EE baseado em técnicas de Desenvolvimento Orientado a Modelos**. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) — Universidade Federal de Pernambuco, 2008.

SOUZA, M. de. **Inteligência Computacional: Notas de aula**. [S.l.]: UDESC, 2017.

TENFEN, R. **Extensão da Abordagem de Desenvolvimento Dirigido a Modelos Para Simulações com Agentes (MDD4ABMS) Para Suportar a Plataforma Repast**. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software) — Universidade do Estado de Santa Catarina (UDESC), 2019.

WOOLDRIDGE, M. **An introduction to multiagent systems**. [S.l.]: John Wiley & Sons, 2009.

WOOLDRIDGE, M. **The Software that Led to the Lockdown**. 2020. Disponível em: <https://cacm.acm.org/blogs/blog-cacm/246511-the-software-that-led-to-the-lockdown/fulltext\#:~:text=One\%20of\%20the\%20key\%20drivers,developed\%20at\%20Imperial\%20College\%2C\%20London>. Acesso em: 20 nov. 2020.

APÊNDICE A – Código Java gerado da classe da doença

O código é referente a doença chamada Micose com o modelo compartimental SIR, em que a progressão do compartimento I é do tipo determinística. A mortalidade é "ao sair do compartimento" com uma taxa de mortalidade de 20% para o agente Humano. A simulação possui dois tipos de agentes, Humano e Pet.

```java
public class Micose {

	private boolean die = false;
	private char states[];
	private char compartmental_Humano;
	private int duration_current_I_Humano = 0;
	private double mortalityRateHumanoI;
	private char compartmental_Pet;
	private int duration_current_I_Pet = 0;

	public Micose() {
		this.mortalityRateHumanoI = 0.2;
		int tamanho = 0;
		tamanho++;
		tamanho++;
		tamanho++;
		states = new char[tamanho];
		int i = 0;
		states[i] = 'S';
		i++;
		states[i] = 'I';
		i++;
		states[i] = 'R';
		i++;
		this.compartmental_Humano = states[0];
		this.compartmental_Pet = states[0];
	}

	public char getStatePosition(int i) {
		return this.states[i];
	}

	public char getCompartmental_Humano() {
		return compartmental_Humano;
	}

	public void setCompartmental_Humano(char compart) {
		this.compartmental_Humano = compart;
	}

	public char getCompartmental_Pet() {
		return compartmental_Pet;
	}

	public void setCompartmental_Pet(char compart) {
		this.compartmental_Pet = compart;
	}

	public void nextCompartimental(Object object) {
		for (int i = 0; i < this.states.length; i++) {
			if (object instanceof Humano) {
				if (this.compartmental_Humano == states[i]) {
					if (i == (this.states.length - 1)) {
						this.compartmental_Humano = this.states[0];
						break;
					} else {
						this.compartmental_Humano = this.states[(i + 1)];
						break;
					}
				}
			}
			if (object instanceof Pet) {
				if (this.compartmental_Pet == states[i]) {
					if (i == (this.states.length - 1)) {
						this.compartmental_Pet = this.states[0];
						break;
					} else {
						this.compartmental_Pet = this.states[(i + 1)];
						break;
					}
				}
			}
		}
		resetDurations(object);
	}

	public void resetDurations(Object object) {
		if (object instanceof Humano) {
			this.duration_current_I_Humano = 0;
		}
		if (object instanceof Pet) {
			this.duration_current_I_Pet = 0;
		}
	}

	public boolean transitionBetweenStates(Object agent) {
```

```
87                  if (agent instanceof Humano) {
88                          getTransitionValueTrigger(this.compartmental_Humano, null, agent);
89                  }
90                  if (agent instanceof Pet) {
91                          getTransitionValueTrigger(this.compartmental_Pet, null, agent);
92                  }
93                  return this.die;
94          }
95
96          public boolean infect(Object agent, Object susceptivel) {
97                  return getTransitionValueTrigger('S', agent, susceptivel);
98          }
99
100         public void contactTransmission(Grid<Object> grid, GridPoint pt,
101         List<Object> objects, Object object, Object agent) {
102                 returnAgents(objects, grid, pt.getX(), pt.getY(), object, agent);
103         }
104
105         public void getHumanoToHumano(Grid<Object> grid, GridPoint pt, int typeOfTransmission,
106         List<Object> listaAgentes, Object agent) {
107                 Humano obj = new Humano();
108                 if (typeOfTransmission <= 0) {
109                         contactTransmission(grid, pt, listaAgentes, obj, agent);
110                 } else {
111                         proximityTransmission(grid, pt, typeOfTransmission, listaAgentes, obj, agent);
112                 }
113         }
114
115         public void getPetToHumano(Grid<Object> grid, GridPoint pt, int typeOfTransmission,
116         List<Object> listaAgentes, Object agent) {
117                 Humano obj = new Humano();
118                 if (typeOfTransmission <= 0) {
119                         contactTransmission(grid, pt, listaAgentes, obj, agent);
120                 } else {
121                         proximityTransmission(grid, pt, typeOfTransmission, listaAgentes, obj, agent);
122                 }
123         }
124
125         public void getPetToPet(Grid<Object> grid, GridPoint pt, int typeOfTransmission,
126         List<Object> listaAgentes, Object agent) {
127                 Pet obj = new Pet();
128                 if (typeOfTransmission <= 0) {
129                         contactTransmission(grid, pt, listaAgentes, obj, agent);
130                 } else {
131                         proximityTransmission(grid, pt, typeOfTransmission, listaAgentes, obj, agent);
132                 }
133         }
134
135         public void getHumanoToPet(Grid<Object> grid, GridPoint pt, int typeOfTransmission,
136         List<Object> listaAgentes, Object agent) {
137                 Pet obj = new Pet();
138                 if (typeOfTransmission <= 0) {
139                         contactTransmission(grid, pt, listaAgentes, obj, agent);
140                 } else {
141                         proximityTransmission(grid, pt, typeOfTransmission, listaAgentes, obj, agent);
142                 }
143         }
144
145         public List<Object> returnAgents(List<Object> objects, Grid<Object> grid, int x, int y,
146         Object object, Object agent) {
147                 for (Object obj : grid.getObjectsAt(x, y)) {
148                         if (!objects.contains(obj)) {
149                                 if (!obj.equals(agent) && (obj.getClass() == object.getClass())) {
150                                         objects.add(obj);
151                                 }
152                         }
153                 }
154                 return objects;
155         }
156
157         public void proximityTransmission(Grid<Object> grid, GridPoint pt, int distancia,
158         List<Object> objects, Object object, Object agent) {
159                 for (int i = -(distancia - 1); i < distancia; i++) {
160                         for (int a = -(distancia - 1); a < distancia; a++) {
161                           int x = returnValueOfEdgeDimension((pt.getX() + i),
162                                                   grid.getDimensions().getWidth());
163                           int y = returnValueOfEdgeDimension((pt.getY() + a),
164                                                   grid.getDimensions().getHeight());
165                           returnAgents(objects, grid, x, y, object, agent);
166                         }
167                 }
168         }
169
170         public int returnValueOfEdgeDimension(int xy, int maxDimension) {
171                 if (xy < 0) {
172                         return 0;
173                 } else {
174                         if (xy > (maxDimension - 1)) {
175                                 return (maxDimension - 1);
176                         }
177                 }
178                 return xy;
179         }
180
```

```
181         public List<Object> getTypeTransmission(Grid<Object> grid, GridPoint pt, Object agent) {
182                 List<Object> listaAgentes = new ArrayList<>();
183                 if (agent instanceof Humano) {
184                         getHumanoToHumano(grid, pt, 2, listaAgentes, agent);
185                 }
186                 if (agent instanceof Pet) {
187                         getPetToHumano(grid, pt, 0, listaAgentes, agent);
188                 }
189                 if (agent instanceof Pet) {
190                         getPetToPet(grid, pt, 2, listaAgentes, agent);
191                 }
192                 if (agent instanceof Humano) {
193                         getHumanoToPet(grid, pt, 0, listaAgentes, agent);
194                 }
195                 return listaAgentes;
196         }
197
198         public boolean getTransitionValueTrigger(char compartimento, Object infected,
199         Object susceptivel) {
200                 if (susceptivel instanceof Humano) {
201                         if ('S' == compartimento && (infected instanceof Humano)){
202                                 if (Math.random() <= 0.6 && !die) {
203                                         return true;
204                                 }
205                         }
206                         if ('S' == compartimento && (infected instanceof Pet)) {
207                                 if (Math.random() <= 0.3 && !die) {
208                                         return true;
209                                 }
210                         }
211                         if ('I' == compartimento) {
212                                 if (duration_current_I_Humano >= 10 && !die) {
213                                         if (Math.random() <= mortalityRateHumanoI) {
214                                                 die = true;
215                                                 return true;
216                                         }
217                                         nextCompartimental(susceptivel);
218                                         return false;
219                                 } else {
220                                         duration_current_I_Humano++;
221                                 }
222                         }
223                 }
224                 if (susceptivel instanceof Pet) {
225                         if ('S' == compartimento && (infected instanceof Pet)) {
226                                 if (Math.random() <= 0.6 && !die) {
227                                         return true;
228                                 }
229                         }
230                         if ('S' == compartimento && (infected instanceof Humano)){
231                                 if (Math.random() <= 0.5 && !die) {
232                                         return true;
233                                 }
234                         }
235                         if ('I' == compartimento) {
236                                 if (duration_current_I_Pet >= 15 && !die) {
237                                         nextCompartimental(susceptivel);
238                                         return false;
239                                 } else {
240                                         duration_current_I_Pet++;
241                                 }
242                         }
243                 }
244                 return false;
245         }
246 }
```

APÊNDICE B – Código Java gerado da classe do agente

Código do agente Humano que possui uma doença chamada Micose. Essa simulação também possui outro agente chamado Pet. Os métodos que não tem relação com a habilidade de propagação de doença foram escondidos, bem como os *getters* e *setters*, para o código ficar mais compacto. Os códigos destacados em amarelo claro são códigos gerados pelo gerador de código desenvolvido por Tenfen (2019).

```
1  package micose;
2  import java.util.ArrayList;
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.Comparator;
6  import repast.simphony.visualizationOGL2D.DefaultStyleOGL2D;
7  import repast.simphony.engine.schedule.Schedule;
8  import repast.simphony.engine.environment.RunEnvironment;
9  import repast.simphony.engine.schedule.ScheduledMethod;
10 import repast.simphony.parameter.Parameters;
11 import repast.simphony.space.grid.Grid;
12 import repast.simphony.space.grid.GridPoint;
13 import repast.simphony.util.ContextUtils;
14 import simphony.util.messages.MessageCenter;
15 import repast.simphony.random.RandomHelper;
16 ppublic class Humano extends DefaultStyleOGL2D {
17   private Micose diseaseMicose;
18   private String compartment_Micose, compartment_Micose_real;
19    private boolean dead;
20    private int count;
21    private Grid<Object> grid;
22
23    public Humano() {
24          this.dead = false;
25          this.count = 0;
26          this.diseaseMicose = new Micose();
27    }
28
29    //getters and setters - códigos escondidos
30
31    @ScheduledMethod(start = 0, interval = 1)
32    public void step() {
33          agentMobilityCapability();
34          if (!dead && RunEnvironment.getInstance().getCurrentSchedule().getTickCount() > 1) {
35                  DiseaseCompartimental();
36          }else {
37                  updateCompartment();
38          }
39    }
40
41    ppublic void die() {
42          //código escondido
43    }
44
45    public void updateCompartment() {
46          switch (this.getDiseaseMicose().getCompartmental_Humano()) {
47                  case 'S' :
48                          this.compartment_Micose_real = "S";
49                          break;
50                  case 'E' :
51                          this.compartment_Micose_real = "E";
52                          break;
53                  case 'I' :
54                          this.compartment_Micose_real = "I";
55                          break;
56                  case 'R' :
57                          this.compartment_Micose_real = "R";
58                          break;
59                  case 'P' :
60                          this.compartment_Micose_real = "P";
61                          break;
62                  default :
63                          this.compartment_Micose_real = "";
64                          break;
65          }
66    }
67
68    private int random(String parameterMin, String parameterMax){
69          //código escondido
70    }
71
```

```
72      private int getMaxMinGridValue(int value, int max) {
73            //código escondido
74      }
75
76      private int getLowerUpperLimitValue(int value, int max, int lower, int upper) {
77            //código escondido
78      }
79
80      private void agentMobilityCapability() {
81       //código escondido
82      }
83
84      private class BestSpot {
85            //código escondido
86      }
87
88    public void DiseaseCompartimental() {
89                  if (diseaseMicose.getCompartmental_Humano() == 'I') {
90                  GridPoint pt = grid.getLocation(this);
91                  List<Object> Objects = this.diseaseMicose.getTypeTransmission(grid, pt, this);
92                  if (Objects.size() > 0) {
93                          for (Object obj : Objects) {
94                                  if (obj instanceof Humano) {
95                                          Humano agent = (Humano) obj;
96                                          if (agent.getDiseaseMicose().getCompartmental_Humano() == 'S') {
97                                                  if (diseaseMicose.infect(this, agent)) {
98                                                          agent.getDiseaseMicose().nextCompartimental(agent);
99                                                          agent.updateCompartment();
100                                                 }
101                                         }
102                                 }
103                                 if (obj instanceof Pet) {
104                                         Pet agent = (Pet) obj;
105                                         if (agent.getDiseaseMicose().getCompartmental_Pet() == 'S') {
106                                                 if (diseaseMicose.infect(this, agent)) {
107                                         agent.getDiseaseMicose().nextCompartimental(agent);
108                                                         agent.updateCompartment();
109                                                 }
110                                         }
111                                 }
112                         }
113                 }
114
115       }
116       if (this.diseaseMicose.getCompartmental_Humano() != this.diseaseMicose.getStatePosition(0)) {
117               if (this.diseaseMicose.transitionBetweenStates(this)) {
118                       die();
119               }
120       }
121       updateCompartment();
122   }
123
124     public int countinfectados() {
125           this.compartment_Micose = "I";
126           if (((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() % 1) == 0)
127                           && (this.compartment_Micose == this.compartment_Micose_real)) {
128                   return 1;
129           }
130           return 0;
131     }
132
133     public int countrecuperados() {
134           this.compartment_Micose = "R";
135           if (((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() % 1) == 0)
136                           && (this.compartment_Micose == this.compartment_Micose_real)) {
137                   return 1;
138           }
139           return 0;
140      }
141
142   public int countsusceptiveis() {
143           this.compartment_Micose = "S";
144           if (((RunEnvironment.getInstance().getCurrentSchedule().getTickCount() % 1) == 0)
145                           && (this.compartment_Micose == this.compartment_Micose_real)) {
146                   return 1;
147           }
148           return 0;
149     }
150 }
```