Universidade Federal do Espírito Santo
Programa de Pós-Graduação em Informática

ProScene: Uma Plataforma para Simulação de Situações

Alessandro Murta Baldi

Vitória - ES, 20 de Dezembro de 2018

Alessandro Murta Baldi

ProScene: Uma Plataforma para Simulação de Situações

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da UFES, como parte dos requisitos necessários para a obtenção do Título de Mestre em Informática.

Universidade Federal do Espírito Santo – UFES

Programa de Pós-Graduação em Informática - PPGI

Orientador: Profa. Dra. Patrícia Dockhorn Costa

Vitória - ES

20 de Dezembro de 2018

Alessandro Murta Baldi

ProScene: Uma Plataforma para Simulação de Situações/ Alessandro Murta Baldi. – Vitória - ES, 20 de Dezembro de 2018-

132 p. : il. (algumas color.) ; 30 cm.

Orientador: Profa. Dra. Patrícia Dockhorn Costa

Dissertação (Mestrado) – Universidade Federal do Espírito Santo – UFES
Programa de Pós-Graduação em Informática - PPGI, 20 de Dezembro de 2018.

1. Situation Awareness. 2. Simulações. I. Patrícia Dockhorn Costa. II. Universidade Federal do Espírito Santo. III. Departamento de Informática. IV. ProScene: uma Plataforma para Simulação de Situações

Aos meus pais, Salvador e Maria de Fátima

Agradecimentos

Agradeço primeiramente a Deus, por me proporcionar a continuação dos meus estudos, terminando mais uma etapa de minha vida.

Agradeço aos meus pais, Salvador e Maria de Fátima pelo apoio, carinho e amor que foram fundamentais em minha jornada acadêmica. O apoio deles foi fundamental para terminar a graduação e não foi diferente com o mestrado.

Agradeço à minha orientadora Profa. Dra. Patrícia Dockhorn Costa, que me orientou nessa jornada e teve uma grande paciência enquanto eu passava por um dos momentos mais difíceis da minha vida.

Ao Prof. Dr. Amaury Antônio de Castro Junior, que novamente fez parte de minha vida acadêmica, vindo do estado de Mato Grosso do Sul e fazendo importantes contribuições ao meu trabalho.

Ao Prof. Dr. João Paulo Andrade Almeida pelas importantes contribuições no trabalho e no artigo.

Apesar de ser difícil citar todos, agradeço aos amigos, demais estudantes, professores e funcionários da universidade, que fizeram parte da minha vida de alguma forma e foram importantes em minha formação acadêmica e convivência profissional.

"It is the theory that describes
what we can observe."
(Albert Einstein)

Resumo

Simulações de sistemas possibilitam que os processos que acontecem na vida real sejam desenvolvidos dentro de um ambiente controlado permitindo experimentações sobre uma ampla gama de condições. Essas experimentações promovem (i) a identificação de possíveis problemas que ocorrem ao longo do tempo em um determinado contexto, (ii) o treinamento de sistemas e organismos simulados e (iii) análises de possíveis estados de sistemas que possam ser alcançados.

Esta dissertação explora um aspecto ainda não encontrado na literatura: a utilização de simulações que considerem explicitamente o conceito de Situação, denominadas neste trabalho de Simulações *Situation-Aware* (SiSA), simulações orientadas a situação. Portanto, as SiSAs visam simular a execução de sistemas baseados em situação, que são sistemas capazes de se adaptar autonomamente às situações, ou seja, reagindo conforme os acontecimentos e promovendo, portanto, uma forma inovadora de *feedback*, mais próxima do que acontece na realidade.

O propósito do trabalho é facilitar o desenvolvimento de SiSAs e, nesse sentido, entrega duas contribuições importantes: (i) uma pesquisa exploratória em diversas ferramentas de simulação comparando os paradigmas de programação, desempenho e características das ferramentas para desenvolvimento de uma SiSA (ii) uma nova plataforma de simulações e situações chamada ProScene.

ProScene é uma plataforma híbrida, possuindo as funcionalidades de uma ferramenta de simulação orientada a agentes e de uma plataforma de gerenciamento de situações. Desta forma, ProScene possui uma característica singular: a de possibilitar que o desenvolvedor implemente situações como agentes da simulação, possibilitando o monitoramento visual das ocorrências de ativação e desativação de situações além de sua localização em relação a simulação.

Palavras-chaves: Situation Awareness. Simulações. Agentes.

Abstract

Systems simulation permits that processes in real life to be developed in a controlled environment allowing experimentation over a wide range of conditions. These experiments promote (i) the identification of possible problems that occur over time in a given context, (ii) the training of simulated systems and organisms and (iii) analyzes of possible states that systems can reach.

This dissertation explores an aspect not yet found in the literature: the use of simulations that explicitly consider the concept of Situation, named in this work as Situation-Aware Simulations (SiSA). Therefore, SiSAs are designed to simulate the execution of situation-based systems, capable of adapting autonomously to the situation of its users, promoting an innovative form of feedback, close to what happens in reality.

The purpose of this work is to facilitate the development of SiSAs and, in this sense, it provides two important contributions: (i) an exploratory research in several simulation tools comparing programming paradigms, performance and characteristics of tools for the development of a SISA, and (ii) new simulations and situations platform called ProScene.

ProScene is a hybrid platform, with the features of an agent-oriented simulation tool and a situation management platform. In this way, ProScene has a unique characteristic: it enables the developer to implement situations as agents of the simulation, allowing the visual monitoring of activation and deactivation of situations in their locations.

Keywords: *Situation Awareness*. Simulations. Agents.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABMS	*Agent-Based Modeling and Simulation*
DRL	*Drools Rule Language*
IDE	*Integrated Development Environment*
LHS	*Left Hand Side*
RHS	*Right Hand Side*
SiSA	*Simulações Situation-Aware*
UFES	Universidade Federal do Espírito Santo

Sumário

1 Introdução

Sistemas baseados em situação são capazes de se adaptar autonomamente à situação de seus usuários promovendo, portanto, interações mais efetivas. São utilizados em diversos domínios e uma das funcionalidades básicas deste tipo de sistema é a capacidade de perceber e reconhecer determinados padrões (também conhecidos como situações) em fatos ocorridos no ambiente no qual está inserido (BALDI et al., 2018) (COSTA et al., 2012).

Um dos primeiros exemplos de sistemas baseados em situação pode ser atribuído ao planejamento de combate aéreo na primeira guerra mundial, levando estratégias de combate por situações aos pilotos. A partir deste trabalho, diversos outros trabalhos utilizando *situation awareness* foram realizados, desde operações espaciais até a medicina (ENDSLEY; GARLAND, 2000).

Um exemplo recente de *situation awareness* é o da febre intermitente como um comportamento que compõe um sistema baseado em situação, na qual há a leitura constante da temperatura de um determinado paciente e a situação febril em determinados momentos, estando o paciente acima da temperatura de 37 graus (PEREIRA; COSTA; ALMEIDA, 2013).

Uma situação é percebida/reconhecida quando um padrão na ocorrência de fatos é satisfeito e a situação deve permanecer ativa enquanto a ocorrência deste padrão for satisfeita. A situação deve deixar de existir quando as condições do padrão deixam de ser satisfeitas(PEREIRA; COSTA; ALMEIDA, 2013). Os sistemas baseados em situação são interessantes para que se possa monitorar um determinado domínio e possibilitar (re)ações de acordo com as situações ocorridas no ambiente.

A partir da implementação do sistema baseado em situação com o domínio do *Aedes aegypti* no mundo real, por exemplo, pode-se ter determinadas (re)ações inteligentes com situações a partir da coleta de dados de sensores em armadilhas. Há a possibilidade de prever e realizar ações de prevenção do mosquito, como a passagem de agentes de saúde e carros fumacê nas regiões em que há a situação crítica de infestação dos mosquitos na armadilha. Por outro lado, elementos conhecidos para a infestação dos mosquitos podem fornecer novas possibilidades ao sistema para o conhecimento das situações: a inclusão de dados do clima, por exemplo, permitiria a construção de situações em relação às chuvas na região e sua ligação com a infestação de *Aedes aegypti*.

A percepção e o conhecimento das ocorrências de situações no ambiente são interessantes em diversos cenários de aplicação, tais como na área da saúde (com a verificação de sensores biométricos para encontrar situações de interesse em doenças) (PEREIRA; COSTA; ALMEIDA, 2013), aviação (situações operacionais do avião) (ENDSLEY, 2017),

controle de tráfego aéreo (situações de localizações, velocidades, comunicações, entre outras de vôo) (ENDSLEY, 2017), sistemas grandes e complexos (situações de operação e parâmetros dos sistemas), sistemas táticos e estratégicos (situações de partes críticas em determinados cenários para tomada de decisão) (ENDSLEY, 2017), dentre outros.

Os sistemas baseados em situação permitem a exploração de diversos domínios com o uso de tecnologias para entender e extrair significados úteis em cada ambiente. Sensores para a coleta dados, dados provenientes de *Big Data* e atuadores para (re)ação nos ambientes são alguns dos exemplos de tecnologias para uso em conjunto aos sistemas baseados em situação. No entanto, essas tecnologias exigem uma experimentação antes da utilização ou execução no mundo real e, dessa forma, a simulação pode ser útil.

Os processos que acontecem na vida real são realizados em velocidades inadequadas para a nossa percepção ou para a extração de informações suficientes com o intuito de se fazer uma extensa análise (ROTH, 2018). Nesse sentido, a simulação de sistemas possibilita que os processos sejam desenvolvidos dentro de um ambiente controlado, fazendo com que existam experimentações sobre uma ampla gama de condições. A experimentação permite a identificação de possíveis problemas que ocorrem ao longo do tempo em um determinado contexto, o treinamento de sistemas e organismos simulados e análises de possíveis estados de sistemas que possam ser alcançados (BALDI et al., 2018).

Existem diferentes tipos de plataformas para simulação, com diferentes objetivos, tais como a simulação de mosquitos *Aedes Aegypti* (BALDI et al., 2017), ecossistemas (REPENNING, 1993), grafos (LARA; VANGHELUWE, 2002) (GHAMARIAN et al., 2012), organismos naturais (GHAMARIAN et al., 2012), organismos artificiais (KLEIN, 2003), comunicações (HELSINGER; THOME; WRIGHT, 2004), áreas urbanas (YANG; LI; ZHAO, 2014) (CARNEIRO et al., 2004), áreas terrestres (CARNEIRO et al., 2004), dentre outros.

Portanto, simulações apoiam o estudo de fenômenos complexos, atividades de planejamento, avaliação de cenários e tomadas de decisão. As ferramentas de simulação auxiliam na modelagem, simulação e estudos de sistemas complexos, sendo úteis no apoio a usuários básicos e avançados para se definir um determinado cenário e avaliá-lo quantitativamente e qualitativamente (FERREIRA, 2017).

Apesar do potencial de simulações para a área de Sistemas baseados em Situação, até o momento da escrita desta dissertação, não foi encontrado um sistema com características de simulação e que apresente uma forma explícita de se programar e observar situações.

Dessa forma, essa dissertação propõe uma nova ferramenta de simulação e verificação de situações chamada *ProScene*. A ferramenta consiste em uma implementação de diferentes tecnologias para construção e execução de simulações além de oferecer funcionalidades de monitoramento e reatividade utilizando-se regras e situações.

1.1 Motivação

Apesar de sensores e atuadores possibilitarem a interação e percepção do mundo real, algumas características dos acontecimentos podem dificultar o processo de entendimento: fatos podem acontecer de uma forma vagarosa ou muito rápida para serem entendidos, podem não ser possíveis de serem experimentados em sua totalidade, podem ser percebidos de forma errônea, podem ser perigosos em determinadas condições, podem acontecer apenas uma vez ou podem ser fatos imperceptíveis a uma simples consulta computacional (ROTH, 2018).

Com essa falta de experimentação, percepção e planejamento na vida real diante de fatos que podem ser úteis no entendimento de um determinado contexto, as simulações podem ajudar a entender como os fatos se desenrolam com o apoio do controle de tempo e espaço. Além disso, simulações podem prever efeitos de determinadas ações ou até mesmo otimizá-las em determinados casos; reproduzir diversas vezes um acontecimento raro na vida real; ajudar a investigar os efeitos de condições difíceis de se encontrar no mundo real e testar fatos de formas diferentes (ROTH, 2018).

Situation-Awareness, por sua vez, faz a percepção dos elementos no ambiente através do volume de tempo e espaço, a compreensão dos significados e a projeção dos acontecimentos no futuro (ENDSLEY, 2017). Eventos e episódios são situações no tempo, cenas são visualmente percebidas como situações, mudanças são sequências de situações e fatos compõem situações (DEVLIN, 2006).

Assim, tanto simulações quanto situações estão relacionadas ao estudo de fenômenos complexos na vida real. Dessa forma, há a motivação pela busca de ferramentas que permitam fazer simulações e utilizem o conceito de situação explicitamente. A construção de uma nova ferramenta também é motivada uma vez que não existem ferramentas com tais requisitos.

1.2 Justificativas

Com o intuito de monitorar situações no decorrer de uma simulação, diversas ferramentas para a elaboração de simulações genéricas foram pesquisadas e, apesar dos benefícios e diferenças que cada uma das ferramentas possui, verificou-se que nenhuma delas apresenta funcionalidade de monitoramento de situações ou o conceito de *Situation-Awareness* para a melhor compreensão do mundo real. Esta característica mostrou-se interessante em diversos cenários reais, tais como a proliferação do *Aedes aegypti* e o trânsito de veículos em grandes cidades.

Desta forma, a inclusão da funcionalidade de monitoramento de situações em uma ferramenta de simulação é justificável a partir do ponto de vista do enriquecimento

contextual de um cenário simulado. Isso motiva a criação de uma ferramenta que possua características para análise de situações simuladas, no qual que este trabalho propõe implementar.

A contribuição da nova ferramenta visa preencher essa lacuna de integração entre simulações e situações, facilitando o desenvolvimento de Sistemas Baseados em Situação e de Sistemas de Simulação. O desenvolvedor de Sistemas Baseados em Situação poderá contar com um ambiente para testes e visualizações de simulações antes de implementá-lo na vida real. O desenvolvedor de Sistemas de Simulação contará com uma ferramenta para analisar situações dentro de suas simulações, possibilitando uma forma nova e inovadora de estudar os resultados das simulações.

1.3 Objetivos

Sabendo-se da necessidade de se implementar uma ferramenta que integre a facilidade de se programar simulações com os métodos de gerenciamento de situações, este trabalho tem como objetivo principal realizar uma pesquisa exploratória com diversas ferramentas de simulação e, a partir desta pesquisa, elaborar uma plataforma que atenda os requisitos citados anteriormente.

De um modo geral, os objetivos da elaboração da nova plataforma são:

- Proporcionar ao programador uma plataforma de simulação única e simples para elaboração de sistemas sensíveis a situações.
- Simplificar a percepção de situações com a verificação visual das situações ocorridas.
- Aumentar o nível de realismo das simulações com o monitoramento de situações que possam ser interessantes ao desenvolvedor, assim como ocorrem na vida real.
- Possibilitar uma verificação visual de situações detectadas, indicando as ocorrências e as localizações de cada uma.
- Agregar funcionalidades às plataformas de simulação e gerenciamento de situações com a implementação de melhorias.
- Oferecer uma plataforma que possibilite uma integração com o mundo real com a modelagem de diversos domínios e, dessa forma, facilite a implantação de um sistema final.

1.4 Metodologia

Inicialmente é realizada uma pesquisa para encontrar ferramentas que suportem explicitamente situações para análise e composição de simulações. Não encontradas ferramentas com esse suporte, há a proposição de uma nova ferramenta de simulação com o suporte a situações.

Diversas ferramentas para a elaboração de simulações genéricas foram pesquisadas em relação aos seguintes critérios: finalidade, tipo de licença (aberta / fechada), linguagem de programação, tipo de análise que a ferramenta faz (análise pela quantidade de estruturas, análise estatística, análise por situações, etc...), facilidade de uso (ao elaborar e executar simulações) e utilização pela comunidade acadêmica. Essa pesquisa serve como um primeiro filtro, permitindo a escolha das melhores ferramentas, que idealmente teriam um código aberto, uma linguagem de programação compatível que possibilite a integração uma com a outra, análise complementar (uma por situações e outra quantitativa), facilidade de uso e boa utilização pela comunidade acadêmica.

Escolhidas as melhores ferramentas na etapa anterior, é elaborada uma pesquisa exploratória utilizando a região da UFES e o *Aedes aegypti* como um vetor de doenças. Tal estudo permite a observação de diversos pontos cruciais para a elaboração de simulações como as características da linguagem de programação na possibilidade de visualização das estruturas e na possibilidade de programação por agentes, o desempenho em tempo / carga de estruturas na memória e possíveis análises que possam ser quantitativas, qualitativas ou em situações e o que a ferramenta permite realizar com simulações.

O resultado do estudo permite a escolha das melhores tecnologias que servirão de base para o desenvolvimento de uma nova ferramenta de simulação com análise de situações. Idealmente, uma nova ferramenta deve ter facilidade no uso, com uma análise complementar da simulação sendo a análise por situações (verificando as ocorrências de situações durante a simulação) e uma analise quantitativa (verificando as estruturas que fazem a composição da simulação).

Após o desenvolvimento da nova plataforma, são realizados estudos de caso para análise da plataforma implementada à luz dos requisitos inicialmente propostos.

1.5 Estrutura da Dissertação

Esta dissertação está estruturada daqui para frente da seguinte forma:

- O segundo capítulo apresenta o referencial teórico, discutindo todas as tecnologias e ferramentas utilizadas no trabalho, bem como uma análise de trabalhos relacionados.

- O terceiro capítulo apresenta uma pesquisa exploratória de ferramentas para simulação, fazendo um levantamento das ferramentas de simulação, comparando-as, analisando-as e, em seguida, escolhendo algumas ferramentas para implementação de um cenário de *Aedes aegypti*. O cenário, então é analisado e comparado em relação a análise que a ferramenta possibilita na simulação, o desempenho da ferramenta e o tipo de programação que a ferramenta possui.

- O quarto capítulo apresenta o projeto e implementação da nova ferramenta *ProScene*, levantando os requisitos e a arquitetura conceitual da ferramenta e apresentando as diretrizes para implementação de simulações utilizando a ferramenta.

- O quinto capítulo discute o projeto e implementação da ferramenta *ProScene*, fazendo um estudo de caso do cenário anteriormente utilizado (*Aedes aegypti*) e de um novo cenário de trânsito na nova ferramenta e, em seguida, analisando-a.

- O sexto capítulo conclui a dissertação, fazendo uma discussão sobre a ferramenta implementada *ProScene* e fazendo um levantamento de possíveis trabalhos futuros.

2 Referencial Teórico e Trabalhos Relacionados

Neste capítulo serão apresentados importantes conceitos e ferramentas utilizados no decorrer da dissertação além de um levantamento dos principais trabalhos relacionados. As simulações com situações, ao qual este trabalho desenvolve, exigem o conhecimento de sensibilidade ao contexto e situações além de plataformas de simulações. Os conceitos serão apresentados nas próximas seções junto a importantes ferramentas para implementação.

2.1 Sensibilidade ao Contexto e Situações

Um dos primeiros conceitos a serem apresentados foi utilizado na ferramenta proposta para a análise de acontecimentos ou contexto.

A definição de contexto, segundo (ABOWD et al., 1999) é a de que contexto é qualquer informação que pode ser usada para caracterizar uma situação pertencente a uma entidade, sendo entidade uma pessoa, um local ou um objeto de um computador. A sensibilidade ao contexto é a capacidade dos sistemas computacionais entenderem os acontecimentos e promoverem informações ou serviços relevantes ao usuário.

A comunicação entre os humanos, feita por meio da conversa, utiliza informações de situação ou contexto implícitas para melhorar a qualidade da conversa. Os humanos podem perceber determinadas ações e acontecimentos durante a conversa para entender uns aos outros. Essa interação não funciona muito bem quando aplicada à comunicação entre humanos e computadores e, desta forma, a sensibilidade ao contexto nos computadores pode ser aplicada para promover a melhoria desta habilidade (ABOWD et al., 1999).

A melhoria do acesso ao contexto pode produzir ferramentas e serviços computacionais úteis. O uso do contexto é muito importante na área de computação ubíqua, na qual o contexto do usuário está sendo modificado rapidamente (ABOWD et al., 1999). Na área de simulações, o contexto da situação pode ser utilizado para uma análise mais real do que está acontecendo na simulação (BALDI et al., 2017), possibilitando ao humano entender o contexto do que ocorre na simulação, ou seja, entendendo o que o computador fornece de informação de uma maneira mais próxima ao que acontece na realidade: através dos acontecimentos na simulação, não apenas números e estatísticas.

A teoria de (ENDSLEY, 2017) para um sistema *Situation Aware* é de que esse tipo de sistema possui características para perceber os elementos do ambiente em razão de um volume de tempo e espaço, a compreensão dos seus significados e a projeção dos

seus acontecimentos em um futuro próximo. Uma situação pode ser definida como uma descoberta de significados das relações entre contextos, com um nome para descrevê-la. O nome dos acontecimentos pode ser chamado de definição da situação, que seria como um humano define uma relação entre os estados na realidade (YE; DOBSON; MCKEEVER, 2011). Assim, uma situação pode ser determinada como uma interpretação semântica de dados, ou seja, as situações são determinadas ocorrências em estruturas e relações (YE; DOBSON; MCKEEVER, 2011).

O mundo não consiste apenas de objetos ou propriedades e relações, mas sim objetos tendo propriedades e relações uns com os outros. Há acontecimentos no mundo, com determinadas propriedades e em determinados objetos, observados através do senso comum pelos seres humanos que são chamados de situações. Eventos e episódios com determinadas propriedades são situações no tempo, visivelmente percebidas (DEVLIN, 2006).

Situações nos ajudam a conceitualizar certas "partes da realidade que podem ser compreendidas como um todo" (HOEHNDORF, 2005) sendo um dos mais fundamentais recursos das entidades no ambiente de computação pervasiva para adaptar dinamicamente o comportamento às mudanças de situação, para cumprir requisitos do usuário, incluindo segurança e privacidade (YAU; LIU, 2006).

Situações são por vezes descritas como um status de um "objeto" (KOKAR; MATHEUS; BACLAWSKI, 2009), o que possibilita identificar situações em fatos e também as propriedades da situação em si.

Um tipo de situação possibilita a consideração de características gerais das situações de um tipo particular, com critérios únicos para identificar situações daquele tipo. A detecção de simulação requer a detecção de propriedades capturadas no tipo de situação das instâncias de entidades. Uma situação é dita ativa enquanto as propriedades de acontecimentos que compõem a situação forem satisfeitas e dita inativa quando essas propriedades deixarem de ser satisfeitas. No caso em que as propriedades deixam de ser satisfeitas, é uma situação passada. O momento em que a situação é detectada é chamado de ponto de ativação de situação e, quando deixa de existir, um ponto de desativação de situação (PEREIRA; COSTA; ALMEIDA, 2013).

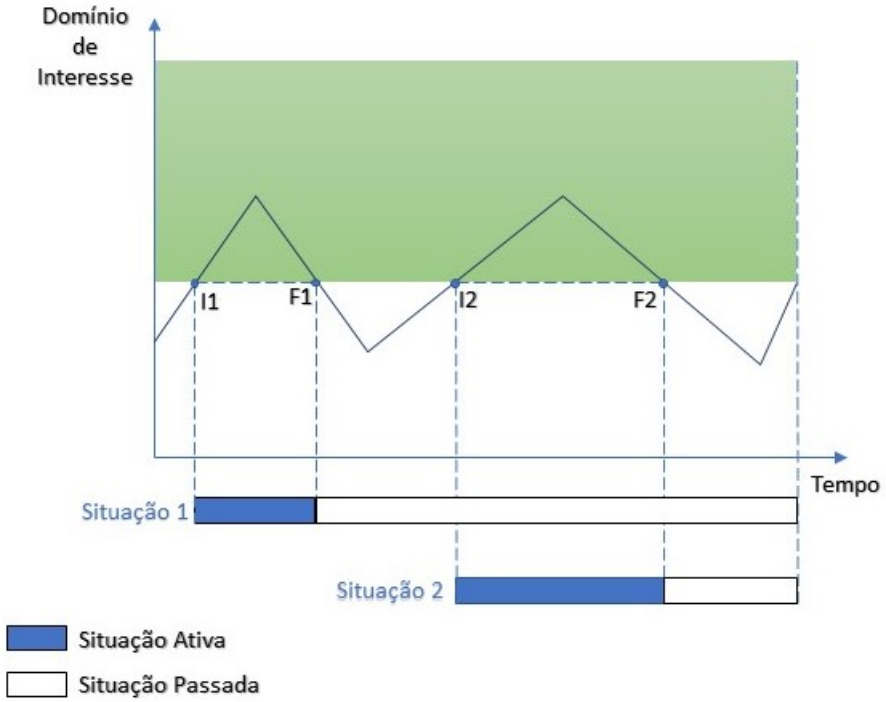

Figura 1 – Exemplo de Situações

A Figura 1 apresenta um ciclo de duas situações sendo detectadas com base no mesmo tipo de situação. A parte vertical apresenta o domínio de interesse da situação detectada, a parte horizontal representa o tempo. Pode-se observar que durante a passagem do tempo, por duas vezes há a satisfação das propriedades de situação, gerando assim, a primeira situação detectada (e ativa) por um período de tempo que começa em I1, em seguida, a desativação em F1. Uma segunda situação é detectada em um outro período de tempo I2 e desativada em F2.

Dessa forma, há certas características de um sistema baseado em situações: os tipos de situação devem ser definidos durante a simulação e detectados durante a execução do programa; os tipos de situação devem ser definidos com referência às entidades que fazem parte das propriedades e relações da situação; propriedades temporais das situações devem ser consideradas (como exemplo o tempo inicial e, para uma situação passada, tempo final e duração da situação ativa).

2.2 Plataformas de Simulação

Um programa simulador é um algoritmo que representa o comportamento de um sistema em razão do tempo, representando um sistema real através de um modelo de precisão com a vantagem de visualização de todo o sistema (CHWIF; MEDINA; SIMULATE, 2015) (FILHO, 2008).

A simulação teve início na década de 60, junto aos primeiros computadores, em

razão do grande número de cálculos matemáticos inviáveis ao cálculo humano. As primeiras utilizações foram feitas na área militar,com o planejamento da distribuição de suprimentos e alocação de recursos (FILHO, 2008).

O programa simulador responde a questões como "o que aconteceria se" e desta maneira economiza recursos e tempo nos estudos dos fenômenos ocorridos. A simulação possibilita o estudo do comportamento e reações através de modelos que imitam em parte ou totalidade os comportamentos em escala menor, permitindo uma manipulação e estudo detalhado (FILHO, 2008).

Como vantagens da simulação, pode-se citar a possibilidade de experimentar determinados comportamentos em exaustão, o controle de condições de experimentos e a descoberta de comportamentos no sistema. Como desvantagens, a simulação demanda recursos e tempo para implementação além de resultados que não podem ser analisados pelo grande volume de dados (CHWIF; MEDINA; SIMULATE, 2015).

Grande parte dos problemas das simulações estão relacionados ao nível inadequado de detalhes da simulação (muito ou pouco detalhamento), interpretações erradas na implementação, documentação incompleta das ferramentas de simulação, *software* complexo, dentre outros (CHWIF; MEDINA; SIMULATE, 2015).

Existem as simulações baseadas em agentes com característica de interagir em um ambiente ou cenário, compartilhado por outros agentes de uma sociedade e que atuam sobre o ambiente, alterando seu estado (BORDINI; VIEIRA, 2003). Cada agente possui um conjunto de capacidades específicas próprios, tendo mecanismos para coordenação e interação das entidades (denominados nessa dissertação como *steps* ou passos a serem seguidos) (BORDINI; VIEIRA, 2003). Os agentes tem como características: ocupar um espaço no ambiente de simulação, autonomia, flexibilidade e heterogeneidade. Os agentes possuem uma forma extensível, flexível e fácil na adição ou remoção de funções (ADAMATTI, 2003).

O agente pode perceber o ambiente (que pode ser físico, interface gráfica ou uma coleção de outros agentes) e, através de ações, atuar sobre este através de um código (RUSSELL; NORVIG, 2016). O código de um agente contém reações que interpretam percepções sobre o ambiente, resolvendo problemas e determinando ações a serem tomadas (TECUCI; DYBALA, 1998).

Diversas aplicações de simulações baseadas em agentes são implementadas de forma a simular alguma situação na realidade. A modelagem de uma simulação em sistemas baseados em agentes envolve a decomposição do fenômeno em conjunto de elementos autônomos, a modelagem de cada um dos elementos em um agente (definindo conhecimento, funções, comportamento e modos de interação), definição do ambiente dos agentes (cenário de simulação) e definição dos agentes com capacidade de ação(ADAMATTI, 2003).

Nas próximas subseções serão apresentados *softwares* de simulação com diversas tecnologias diferentes. O *Groove* é um *software* com características de simulação que utiliza a reescrita de grafos. *RePast* já utiliza a simulação por agentes. Em seguida, haverá as seções do *Processing* e *Scene* em que há a apresentação de outros dois *softwares* de simulação com características próprias.

2.2.1 *Groove*

Groove, ou *GRaphs for Object-Oriented VErification*, é uma ferramenta de simulação e verificação de sistemas através da reescrita de grafos. A ferramenta foi implementada e é mantida pela Universidade de Twente. O *Groove* é um tipo de ferramenta que faz uso de linguagem visual através da manipulação de grafos, formados por nós e arestas.

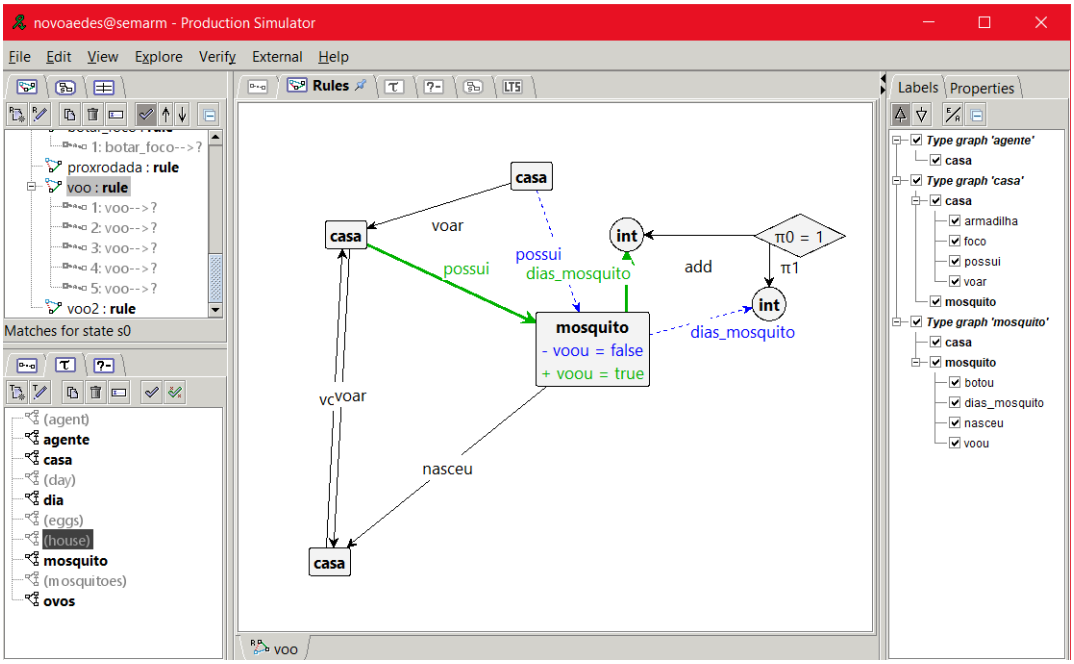

Figura 2 – Ambiente de Desenvolvimento do *Groove*

Para se realizar uma implementação no *Groove*, é preciso implementar um cenário inicial utilizando arestas e nós através do ambiente de desenvolvimento (Figura 2). As arestas são anotadas com rótulos para identificação e caracterização de relacionamentos entre dois nós conectados. Pode-se ter também adicionalmente atributos (como valores inteiros, booleanos, etc) que descrevem certas propriedades de cada nó.

O cenário inicial de arestas e grafos é então transformado pelo *Groove* por regras de transformações de grafos. Uma regra é também um grafo, com o *LHS - Left Hand Side* e o *RHS - Right Hand Side* em um único grafo, sendo o *LHS* uma precondição para a aplicação da regra e o *RHS* o efeito da aplicação. A reescrita é aplicada ao cenário de grafos, que inicia em um grafo inicial (cenário inicial) e são feitas, então, transformações a partir desse cenário que levam a um novo estado.

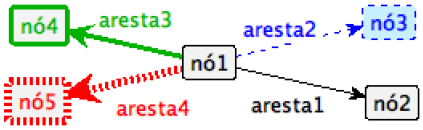

Figura 3 – Sintaxe e Semântica de Regras de Reescrita no *Groove*

A Figura 3 apresenta a sintaxe e semântica do *Groove* na aplicação de regras de transformação de elementos. O papel dos nós e arestas em uma regra é identificado pelo tipo de tracejado na aresta e no entorno do nó. Elementos em tracejado fino (o nó indicado por "nó3" e a aresta com rótulo "aresta2" na Figura 3) são elementos que devem estar presentes no estado atual da simulação e que são removidos quando a regra é aplicada. De forma similar, elementos em traço comum fino ("nó1", "nó2" e "aresta1") também são elementos que devem estar presentes no estado atual mas estes são preservados pela aplicação da regra. Elementos em tracejado grosso ("nó5" e "aresta4") não podem ocorrer no estado atual, caso contrário a regra não é aplicável. Finalmente, elementos em traço comum grosso ("nó4" e "aresta3") são criados pela regra.

Uma simulação no *Groove* essencialmente é modelada através de grafos como entidades e domínio da simulação. As arestas dos grafos representam as relações das entidades umas com as outras. As regras de reescrita essencialmente modificam as estruturas de grafos e suas relações, representando o comportamento da simulação. A execução da simulação no *Groove* se dá através da aplicação das regras de reescrita nos grafos iniciais, dessa forma, modificando-os para um novo cenário e possibilitando a visualização dos acontecimentos em grafos. O cenário final, em grafos, é o resultado do cenário de simulação e pode ser analisado tanto pela visualização quanto pelo número de estruturas.

2.2.2 *Repast*

RePast (*Recursive Porous Agent Simulation Toolkit*) é uma ferramenta que usa a modelagem baseada em agentes e simulação (ABMS - *Agent-Based Modeling and Simulation*), em que os agentes possuem comportamento independente e autônomo. *Repast* suporta diversas linguagens, como o *Logo* (*ReLogo*), *Java* e *C#*. Utilizou-se nesta dissertação o *ReLogo* em razão da recomendação de vários artigos do *RePast* uma vez que é uma linguagem simples e rápida para implementação de simulações (BALDI et al., 2017).

O *software* é gratuito e de código aberto, desenvolvido por David Sallach, Nick Collier, Tom Howe, Michael North e outros na Universidade de Chicago com a colaboração do Laboratório Nacional Argonne, posteriormente sendo mantido pela *Repast Organization for Architecture and Design* (*ROAD*), um grupo voluntário formado por diversas organizações industriais, governamentais e acadêmicas. A comunidade do *RePast* é grande e continua crescendo(NORTH; COLLIER; VOS, 2006).

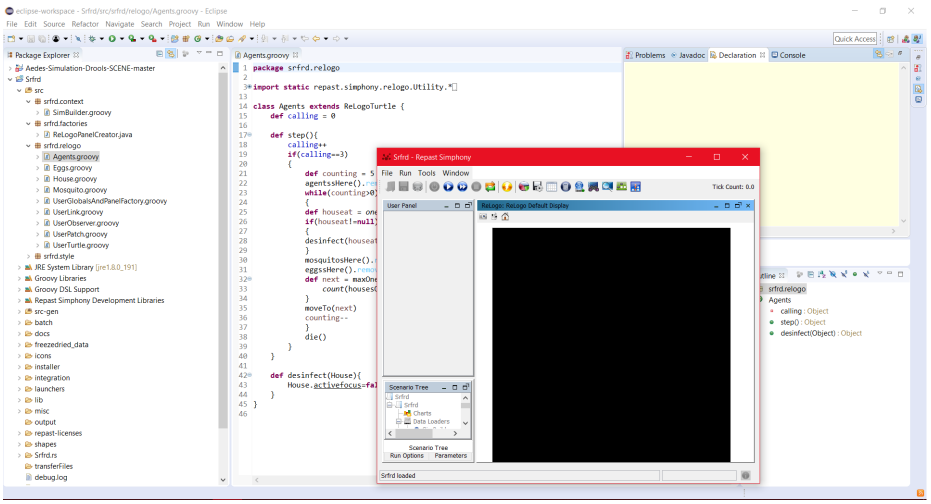

Figura 4 – Ambiente de Desenvolvimento *ReLogo*

Na Figura 4 pode-se verificar o ambiente *RePast* (*ReLogo*) para desenvolvimento de simulações. O *RePast* pode ser utilizado em uma grande quantidade de aplicações para simulação, como sistemas sociais, sistemas evolucionários, modelagem de mercado, análise industrial, dentre outros (NORTH; COLLIER; VOS, 2006).

Essencialmente, para se modelar uma simulação em *RePast* é necessário criar os agentes (entidades) participantes da simulação, definir os *steps* (comportamento autônomo) que cada agente possui e definir as formas e cores que cada agente deve assumir na visualização da simulação.

Cada agente da simulação é autônomo em seu comportamento, utilizando os *steps* para se relacionar com o ambiente, outros agentes ou modificar atributos próprios. Inicialmente deve-se adicionar agentes em determinadas localizações do ambiente de

simulação (cenário inicial) e, com a execução da simulação, os agentes dispostos na simulação assumem novas formas (de localização, quantidade, outros agentes, etc...) e a simulação poderá ser analisada. A análise da simulação do *RePast* pode ser realizada pela visualização dos agentes no ambiente de execução ou com a utilização das ferramentas de análise estatística próprias do *RePast*.

2.3 *Processing*

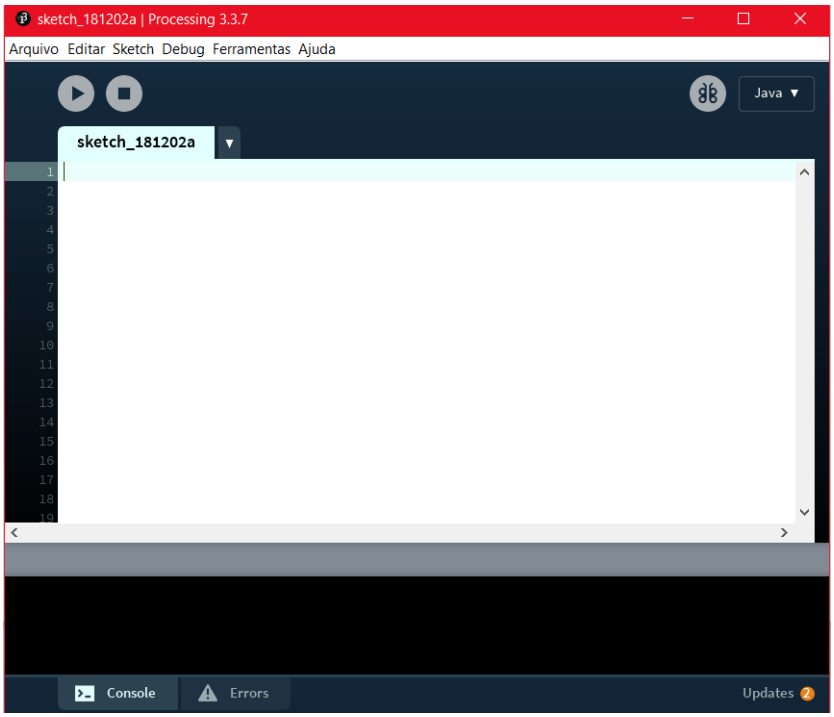

Figura 5 – Ambiente de Desenvolvimento *Processing*

Processing é uma linguagem de programação de alto nível e um ambiente de desenvolvimento (pronto para uso), chamado de "*flexible sketchbook*" (Figura 5). Foi desenvolvido em 2001 com o objetivo de promover o ensino de programação com artes visuais e inclusão da tecnologia nas artes. Hoje, o *Processing* é uma ferramenta de propósito genérico e é utilizado por estudantes, artistas, designers e pesquisadores para aprender a programar e prototipar *softwares* (FRY; REAS, 2018).

O *Processing* é gratuito e de código aberto, possibilita a interação de programas em 2D e 3D através da integração com o *OpenGL*. Os *softwares* produzidos pelo *Processing* podem ser rodados em computadores com *Linux*, *Mac OS X*, *Windows*; Celulares *Android* e dispositivos embarcados *ARM* (FRY; REAS, 2018).

Alguns exemplos de *software* produzidos pelo *Processing* são aplicativos de celular (como o 2+ Dengue (BALDI et al., 2015)), *software* de ensino adaptativo (como proposto por (BALDI, 2016)), *website* de visualização e exploração de genes *microRNA* ((BETEL et

al., 2008)), visualização de dados (como o experimento em física (COLUCI et al., 2013)), ferramenta de simulação baseada em eventos discretos (ferramenta *SimPack* (ALLAN, 2010)), sistemas simuladores de partículas (como proposto em (KILIAN; OCHSENDORF, 2005)), dentre outros.

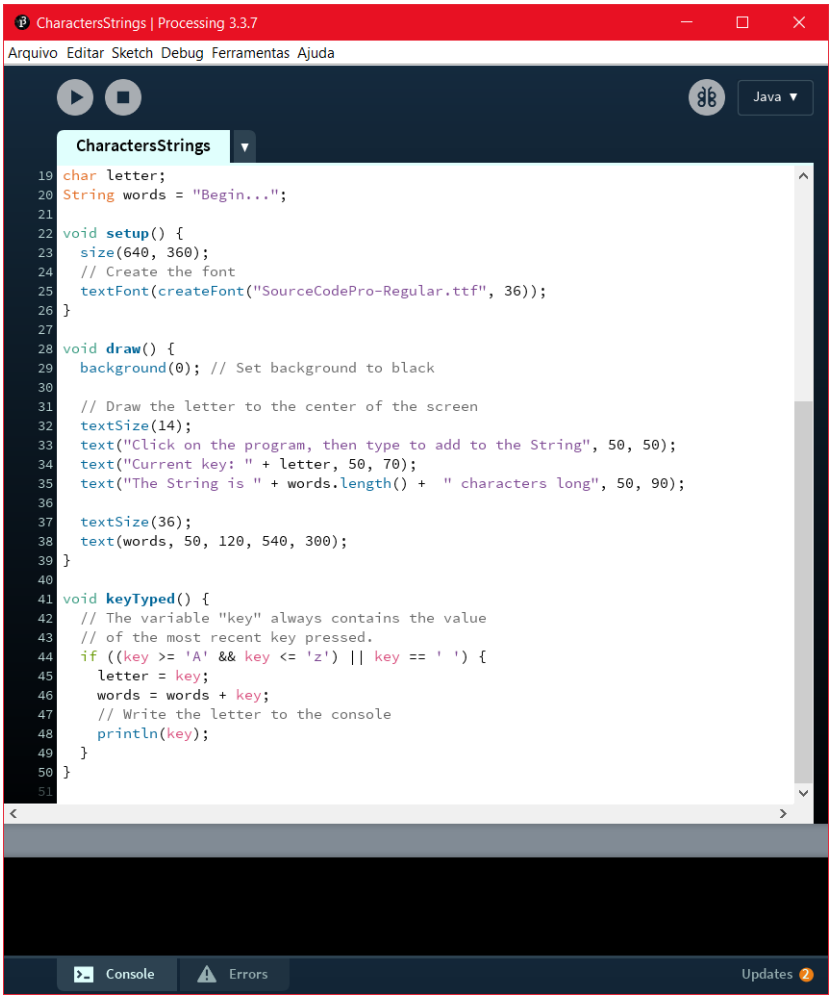

Figura 6 – Exemplo de Desenvolvimento no *Processing*

O diferencial do *Processing* está na facilidade de programá-lo. Como pode ser visto na Figura 6, há dois métodos principais: *Setup* e *Draw*. *Setup* é o método que é realizado uma única vez, para conFiguração do ambiente de trabalho, como por exemplo a inserção de um *background* uma única vez. *Draw* é o método principal, executado enquanto o programa estiver ativo. O .*Draw* possui também a característica de ser executado como *loop*, de forma indefinida. Também, tudo que é colocado em *Draw* aparece na interface do *Processing* depois de toda a execução do código, ou seja, os objetos visuais aparecem quando a execução do código retorna ao ponto inicial do *Draw*. Uma outra característica do *Processing* é que as formas e Figuras que aparecem ao decorrer do código sobrepõem umas às outras, ou seja, o código do último desenho sobrepõe os anteriores. Também, o *Processing* mantém todas as variáveis e tipos de forma global (*static*), para que sejam

acessadas pelas classes independente de quais forem (no exemplo da Figura 6, os tipos e classes são o *char* "*letter*" e a *String* "*words*". Fazendo uma comparação ao *Java*, é como se todas as classes estivessem na *Main*, acessando as variáveis e tipos diretamente.

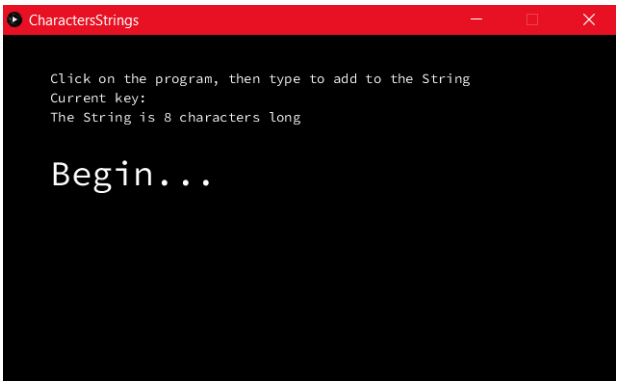

Figura 7 – Resultado do Exemplo de Desenvolvimento

O exemplo em questão (Figura 6) cria duas variáveis ("*letter*" e "*words*"). Em seguinda, através do *Setup*, faz a conFiguração do tamanho do ambiente de visualização de 640 pixels em x e 360 pixels em y (*size(640, 360)*) além da fonte a ser utilizada e tamanho (*textFont(createFont("SourceCodePro-Regular.ttf", 36))*). Em *draw*, o fundo do ambiente é conFigurado em preto (*background(0)*), tamanho do texto em 14, em seguida há a impressão dos textos na tela, sendo o primeiro a partir da posição X 50 e Y 50 *(text("Click on the program, then type to add to the String", 50, 50)*), o segundo texto faz a concatenação entre o texto e o conteúdo da variável "*letter*" em X 50 e Y 70 (*text("Current key: "+ letter, 50, 70)*), o terceiro texto faz a concatenação entre um texto, o tamanho do texto na variável "*words*" e outro texto, começando na posição X 50 e Y 90 (*text("The String is "+ words.length() + "characters long", 50, 90)*). Após isso, há a mudança do tamanho de fonte para 36 (*textSize(36)*) e em seguida um novo texto a ser exibido utilizando a variável *"words"* em uma caixa de texto com X iniciando em 50, Y em 120 e com um tamanho de X de 540 e Y de 300 (*text(words, 50, 120, 540, 300)*).

Também, no exemplo da Figura 6, há um método de interação com o teclado (*keytyped*). Toda vez que uma tecla é pressionada, há a chamada do método. Dentro do método, há um comparativo se a tecla pressionada está entre A e Z (alfabeto) ou se é um espaço *(if ((key >= 'A' && key <= 'z') || key == ' '))*. Se o comparativo for satisfeito, a tecla é atribuída a variável "*letter*" (*letter = key*) e concatenada a variável "*words*" (*words = words + key*). Em seguida, há a impressão da tecla na *IDE* do *Processing* (*println(key)*). O exemplo sendo executado encontra-se na Figura 7.

Outro recurso do *Processing* é a integração com mais de 100 bibliotecas, que estendem a sua funcionalidade padrão. A documentação sobre a ferramenta é muito bem desenvolvida em sua página principal e há vários livros disponíveis para aprendizado da linguagem (FRY; REAS, 2018).

Pela versatilidade, integração com diferentes arquiteturas e plataformas, o *Processing* é utilizado para diversas finalidades, incluindo a simulação computacional.

Uma simulação no *Processing* envolve primeiramente uma escolha de abordagem de simulação. Por ser uma linguagem de programação de uso geral, o *Processing* apresenta diversas maneiras de se realizar uma simulação, sendo algumas delas: simulação 3D, simulação Celular e Simulação de Fenômenos Físicos. A forma de simulação voltada a agentes, assim como o *RePast*, é possível de ser implementada no *Processing*. Dessa forma, entidades (agentes) possuem um comportamento autônomo na simulação, definido por *steps*, ou passos, fazendo com que os agentes interajam entre si e entre o ambiente durante cada processo de *step*.

2.4 *Scene*

Scene é uma plataforma que suporta nativamente *situation-awareness* através do desenvolvimento de regras para a verificação de padrões e, consequentemente, a detecção de situações (COSTA et al., 2016).

Foi implementada sobre o *JBoss Drools* (BROWNE, 2009) e estende o suporte a nativamente suportar regras de situação (*Situation Awareness*) com a especificação do ciclo de vida de uma situação como um padrão em uma regra (PEREIRA; COSTA; ALMEIDA, 2013). *Scene* foi desenvolvido por Isaac Pereira em 2013 na Universidade Federal do Espírito Santo e desde então é mantido em seu repositório com atualizações periódicas para acompanhar o desenvolvimento do *Drools*, parte integrante do *Scene*.

A plataforma *Scene* possibilita a especificação de tipos de situação em estruturas e comportamentos que são realizados por classes e regras, respectivamente. As classes que definem tipos de situação devem ser especificadas com base em uma classe abstrata *Scene* chamada de "*Situation Type*" que, por sua vez, exibe as características de serem definidas em razão de fatos de situação (que devem existir, ou seja, estarem ativos) de acordo com uma pré-definição de existência definida pelas regras. As regras de situação são necessárias para especificar "quando e como" a situação será encontrada. A parte de comportamento da situação é caracterizada por padrões condicionais (*LHS*) da declaração de situação e, dessa forma, a situação definida nas classes é ativada (enquanto o padrão for satisfeito) ou desativada (enquanto o padrão não for satisfeito) (PEREIRA; COSTA; ALMEIDA, 2013).

```
declare Fever extends Situation
    febrile: Person @part
end
```

Figura 8 – Declaração de Situação no Scene

Um exemplo de especificação de uma parte estrutural de um tipo de situação pode ser visto na Figura 8. Esse é um exemplo presente em (PEREIRA; COSTA; ALMEIDA,

2013), no entanto, atualizado para as versões mais recentes do *Scene*. O exemplo utiliza um cenário de febre no qual uma pessoa com 37 graus ou mais pode estar numa situação febril *(Fever Situation Type)*. Na Figura, há a especificação do tipo de situação e suas propriedades. Há a declaração do tipo de situação febril (*Fever*) e a especificação da entidade (pessoa) que faz parte da situação (*febrile: Person*) utilizando a notação "*@part*".

As regras de especificação *Scene* são definidas através de uma linguagem específica chamada *Drools Rule Language* (*drl*). Uma declaração de regra no *Scene* consiste de uma pré-condição (*LHS*) e uma consequência quando a condição for encontrada (*RHS*). Uma regra consiste em um conjunto de condições definidas quando a pré-condição é existente (*LHS*) e um conjunto de ações a serem executadas (*RHS*) (PEREIRA; COSTA; ALMEIDA, 2013).

O *LHS* consiste em diversos elementos condicionais, que podem ser combinados como operadores lógicos, como "*and*", "*or*", "*not*" e "*exists*" e operadores de relações como "contains" e "member of" (PEREIRA; COSTA; ALMEIDA, 2013). Já *RHS* consiste de código procedural (em *Java*) a ser executado quando as condições no *LHS* forem satisfeitas (PEREIRA; COSTA; ALMEIDA, 2013). A ferramenta contém um algoritmo de alto desempenho chamado *PHREAK*, uma evolução do algoritmo *RETE*, para fazer a avaliação das regras e achar os padrões em fatos na *Working Memory* (PEREIRA; COSTA; ALMEIDA, 2013).

```
rule "Fever"
    @role(situation)
    @type(Fever)
    When
        $febrile: Person(temperature > 37)
    then
        SituationHelper.situationDetected(drools)
end
```

Figura 9 – Especificação de Regra no Scene

Na Figura 9, observa-se o exemplo de especificação da parte comportamental do cenário febril apresentado por (PEREIRA; COSTA; ALMEIDA, 2013). A regra de situação utiliza a notação de situação (*@role(situation)*) para indicar que a regra especificada na parte do *Drools* (*DRL*) é de situação e qual o tipo de situação (*@type(fever)*) ao qual a regra pertence. Na parte *LHS* da regra (*when*), é atribuído ao elemento *$febrile* uma pessoa que possui temperatura maior que 37 (*$febrile: Person(temperature > 37)*). Quando a regra é satisfeita, então, a parte *RHS* da regra (*then*) é feita e uma invocação a *API* procedural do *Scene* é realizada (*SituationHelper.situationDetected(drools)*).

Retornando à Figura 1, pode-se considerar que a temperatura do paciente (medida por um sensor, por exemplo) como a linha que percorre o tempo e o domínio de interesse. Em um determinado momento (I1) há a ativação da situação febril e esta permanece enquanto a temperatura estiver maior que 37 graus (domínio de interesse). Em seguida,

ocorre uma desativação da situação (F1) e esta permanece inativa. Uma outra situação ocorre (I2) e permanece ativa até um determinado período (F2).

O *Drools*, parte integrante do *Scene*, suporta nativamente operadores para relacionar eventos em uma perspectiva temporal. Todos os treze operadores lógicos presentes em (ALLEN, 1990) estão disponíveis, incluindo a negação. É possível, por exemplo, definir condições para que um evento aconteça antes de outro ou para que dois eventos sejam sobrepostos no tempo. Os eventos no *Drools* são tratados como ocorrências no passado, ou seja, há determinados momentos em que as propriedades dos eventos são capturadas e, com isso, os eventos são acionados em determinados instantes. Do ponto de vista de situações, o *Drools* não trabalha com eventos "ativos", ou seja, não há a especificação de eventos que possuem um tempo determinado, não sendo possível de se trabalhar com situações. Dessa forma, o *Scene* enriquece a funcionalidade do *Drools* suportando operadores temporais para o controle de eventos e situações.

2.5 Trabalhos Relacionados

A seguir serão apresentados critérios para discussão, discussão inicial, tabela comparativa e comparação de trabalhos relacionados.

2.5.1 Critérios para Discussão

Os critérios para discussão são levantados e comparados de acordo com os objetivos e requisitos do trabalho proposto, descritos na sessão 1.3.

Como critérios para discussão dos trabalhos relacionados, pode-se destacar a caracterização dos trabalhos como uma plataforma de simulação e situação, possibilitando a simulação de acontecimentos como situações ao decorrer da implementação. Esse critério é importante pois apresenta relação com o objetivo do trabalho de "Proporcionar ao programador uma plataforma de simulação única e simples para elaboração de sistemas sensíveis a situação".

Também, um outro critério é a possibilidade de se trabalhar explicitamente com situações ao decorrer da simulação. Dessa forma, situações são implementadas explicitamente no código e reconhecidas, sendo importantes para a captura de determinados padrões de acontecimentos, ou situações. Dessa forma, cumpre o requisito de "Oferecer uma plataforma que possibilite uma integração com o mundo real com a modelagem de diversos domínios e, dessa forma, facilite a implantação de um sistema final". Isso é, fornecer ao programador uma forma explícita de capturar ocorrências de situações, possibilitando a captura de acontecimentos como o mundo real.

A análise visual de situações ocorridas é um critério para discussão entre os trabalhos

relacionados, indicando os momentos em que há a ocorrência de situações através de ícones, cores e desenhos. Isso cumpre o critério de "Possibilitar uma verificação visual de situações detectadas, indicando as ocorrências e as localizações de cada uma".

Um outro critério é a característica de propósito geral da ferramenta proposta, possibilitando a implementação de diferentes cenários. Isso cumpre o objetivo de se implementar uma ferramenta com propósito geral, sem um cenário específico para trabalho, possibilitando o uso em diferentes propostas de pesquisa.

2.5.2 Discussão Inicial

Alguns trabalhos relacionados a sistemas baseados em situação são propostos por (ANGELOPOULOU; MYKONIATIS; KARWOWSKI, 2015), (MANCUSO et al., 2012), (ÖZYURT; DÖRING; FLEMISCH, 2013), (JUAREZ-ESPINOSA; GONZALEZ, 2004) e (WRIGHT; TAEKMAN; ENDSLEY, 2004). Os trabalhos descrevem sistemas que simulam determinados cenários ou auxiliam o usuário na tomada de decisões usando noção de situação.

Em "A framework for simulation-based task analysis-The development of a universal task analysis simulation model" (ANGELOPOULOU; MYKONIATIS; KARWOWSKI, 2015), o autor propõe um novo *framework* e modelo de simulação de tarefas humanas como uma ferramenta de decisão, fazendo a simulação de indivíduos com uma série de características, estimando a carga de trabalho e erros humanos. Com a ferramenta é possível analisar e predizer o comportamento de trabalhadores em determinadas tarefas em razão do tempo e das probabilidades de conseguirem desempenhar as tarefas. Assim como o trabalho proposto nesta dissertação, o *framework* realiza a simulação de organismos vivos, no entanto, utiliza a linguagem UML (*Unified Modeling Language*) na modelagem da simulação enquanto nesta dissertação é proposta a modelagem por uma linguagem de programação ou linguagem visual. Apesar de trabalhar com os acontecimentos ocorridos durante a simulação, o *framework* não possui uma forma de definir e analisar situações de uma forma explícita.

"idsNETS: An experimental platform to study situation awareness for intrusion detection analysts" (MANCUSO et al., 2012) propõe um sistema de simulação e análise de *cyber* segurança. O sistema implementa um sistema de detecção de invasões com o aspecto de análise por situação para detectar, extrair e estabelecer informações importantes a partir de uma simulação na ferramenta *NeoCITIES Experimental Task Simulator (NETs)*. A partir do *NETs*, os autores criaram a ferramenta *idsNETS*, com vários mecanismos que permitem o estudo de *situation-awareness* na escala reduzida da simulação. O sistema de simulação e análise de *cyber* segurança se aproxima do objetivo desta dissertação, apresentando um sistema de simulação com aspectos de *situation-awareness*, no entanto, as simulações realizadas no sistema não são de propósito geral, sendo específicas ao cenário

de cyber-segurança.

Um outro trabalho relacionado é "Simulation-based development of a cognitive assistance system for navy ships" (ÖZYURT; DÖRING; FLEMISCH, 2013) em que o objetivo é desenvolver um estudo através da modelagem de um sistema complexo e cognitivo de assistência (*cognitive assistance system - COGAS*) para aplicações militares da marinha, com as características para a tomada de decisão de um sistema de comando e controle. O sistema, então, é testado em um simulador, verificando se há a percepção das situações de interesse. O trabalho proposto nesta dissertação se assemelha na percepção de situações que o sistema possui ao decorrer da simulação, no entanto, há a possibilidade de trabalhar com diferentes tipos de cenários na simulação, o que não ocorre com o sistema proposto por (ÖZYURT; DÖRING; FLEMISCH, 2013), sendo voltado apenas a aplicações de combate.

Assim como "Simulation-based development of a cognitive assistance system for navy ships" (ÖZYURT; DÖRING; FLEMISCH, 2013), "Situation awareness of commanders: a cognitive model" (JUAREZ-ESPINOSA; GONZALEZ, 2004) utiliza um modelo computacional baseado em *situation-awareness* para o comando militar e controle de operações de batalha através da tomada de decisão militar (*Military decision making - MDM*). Também, apesar do modelo computacional levar em conta situações na análise dos cenários de batalha, há um cenário bastante específico, diferenciando-se do trabalho proposto pela dissertação de ser uma ferramenta de uso geral para simulações.

"Objective measures of situation awareness in a simulated medical environment" (WRIGHT; TAEKMAN; ENDSLEY, 2004) propõe a inclusão de *Situation Awareness* em simulações com pacientes humanos. O trabalho propõe um sistema baseado em situação para o treinamento e avaliação prática de médicos anestesiologistas na percepção de situações, possibilitando aos médicos em treinamento que tenham um *feedback* do desempenho durante o processo de simulação. Assim como o trabalho proposto nessa dissertação, o feedback de desempenho da simulação proposto por (WRIGHT; TAEKMAN; ENDSLEY, 2004) utiliza-se de situações na percepção dos acontecimentos, sendo assim, a análise por situação em simulação é uma semelhança. No entanto, a proposta da dissertação diferencia-se no conceito de realizar análises em cenários simulados de qualquer área, não apenas na avaliação de médicos anestesiologistas.

O trabalho "Simulação de Aplicação de Armadilhas no Combate ao Aedes aegypti" (BALDI et al., 2017) apresenta uma simulação em grafos do *Aedes aegypti* e de armadilhas utilizando o *software Groove* na região da UFES com dados convertidos em grafos a partir de informações do serviço *OpenStreetMap*. A simulação leva em conta os aspectos comportamentais do mosquito e do ambiente para a execução. Em relação ao trabalho da dissertação, a simulação do *Aedes aegypti* proposta por (BALDI et al., 2017) apresenta o mesmo cenário para estudos, no entanto, foram utilizadas outras ferramentas na dissertação para uma comparação. Também, não há aspectos explícitos de situação analisados no

cenário de (BALDI et al., 2017).

O trabalho "Multi-agent modeling and simulation of an Aedes aegypti mosquito population" (ALMEIDA et al., 2010) faz uso de informações a respeito do *Aedes aegypti* e sua população na cidade de Belo Horizonte para a simulação e modelagem, além de verificar o resultado da simulação, comparando-o com dados reais. É extremamente relevante ao tema da pesquisa, dado que possui o foco de verificar a população do *Aedes aegypti* e utilizar o *Repast* para modelar os diversos agentes de simulação. Além disso, o trabalho também utiliza as armadilhas no combate ao *Aedes aegypti* para a verificação dos acontecimentos ao decorrer da simulação. Entretanto, o trabalho não faz o uso de situações na análise dos acontecimentos como a proposta da dissertação.

Também, a respeito do *Aedes aegypti*, pode-se destacar o trabalho "DengueME: um framework de software para modelagem da dengue e seu vetor" (FERREIRA, 2017), em que é uma ferramenta implementada a partir do *software* de modelagem e simulação *TerraME*. A ferramenta implementada é de propósito específico e possibilita uma análise do *Aedes aegypti* em razão do ambiente, população, características do mosquito, clima, dentre outras características espaciais e temporais, também levando em conta a transmissão do vírus da dengue. Apesar de ser uma ferramenta que possibilita a modelagem de um cenário do *Aedes aegypti*, não faz a análise por situações e não permite a realização de simulações com outros tipos de cenário, como proposto por essa dissertação.

Em relação ao comportamento das armadilhas para o *Aedes aegypti*, pode-se destacar o trabalho "2+ Dengue" (BALDI et al., 2015) em que há a criação de um aplicativo utilizando o *software Processing* com instruções para a elaboração de armadilhas da Dengue utilizando garrafas PET. Também, o aplicativo ensina a respeito da armadilha e faz a coleta de dados da localização e da armadilha ao interagir com o usuário ao decorrer do tempo. A proposta da dissertação de simulação, no entanto, é outra.

Em relação ao cenário de trânsito, destaca-se o trabalho "Traffic simulation using agent-based models" (LJUBOVIć, 2009) que se utiliza de dados de tráfego para a realização de uma simulação no *software NetLogo* verificando os efeitos de congestionamento em rodovias. O trabalho possui um levantamento dos comportamentos de uma simulação baseada em agentes para a implementação de simulações de trânsito. Apesar da dissertação basear em alguns aspectos de modelagem de comportamento de trânsito deste trabalho, diferentemente do proposto pela dissertação, o trabalho não faz uso de situações na análise da simulação.

Um outro trabalho é "Agent-based simulation framework for mixed traffic of cars, pedestrians and trams" (FUJII; UCHIDA; YOSHIMURA, 2017) que propõe um *framework* para a realização de simulações baseadas em agentes com a interação entre carros, pedestres e bondes. Diferentemente do proposto pela dissertação, a ferramenta não possibilita a análise de situações nos cenários simulados e também modela apenas cenários de trânsito,

sendo de propósito específico.

Nas próximas seções, serão apresentadas discussões comparando os trabalhos encontrados com o requisito do trabalho proposto de forma mais detalhada. Primeiramente serão apresentados os critérios para discussão, em seguida haverá uma tabela comparativa e a finalização será a discussão da comparação dos trabalhos.

2.5.3 Tabela Comparativa

A tabela 1 apresenta um panorama dos trabalhos relacionados à luz dos critérios levantados na seção 2.5.1.

No eixo vertical da tabela são apresentados os trabalhos (T1 a T11) e suas características em relação aos critérios.

No eixo horizontal são apresentados os critérios de cada ferramenta, sendo eles: apresentar-se como uma plataforma de simulação e situação, trabalhar com situações de forma explícita, possibilidade de analisar visualmente as situações e característica de utilização em diferentes cenários através do propósito geral.

O conteúdo "Sim" indica que possui a característica apresentada. "Parcial"'indica que possui partes da característica. "Não" indica que não possui a característica.

Tabela 1 – Tabela Comparativa de Critérios

	Plataforma de Simulação e Situação	**Situações Explícitas**	**Análise Visual de Situações**	**Propósito Geral**
T1	Sim	Não	Parcial	Não
T2	Sim	Sim	Sim	Não
T3	Parcial	Sim	Sim	Não
T4	Parcial	Sim	Sim	Não
T5	Não	Sim	Parcial	Não
T6	Não	Não	Não	Sim
T7	Não	Não	Não	Sim
T8	Não	Não	Não	Não
T9	Não	Não	Não	Não
T10	Não	Não	Não	Sim
T11	Não	Não	Não	Não

Os trabalhos apresentados na Tabela 1 (T1 a T11) são definidos pelos seguintes nomes:

T1: A framework for simulation-based task analysis-The development of a universal task analysis simulation model

T2: idsNETS: An experimental platform to study situation awareness for intrusion detection analysts

T3: Simulation-based development of a cognitive assistance system for navy ships

T4: Situation awareness of commanders: a cognitive model

T5: Objective measures of situation awareness in a simulated medical environment

T6: Simulação de Aplicação de Armadilhas no Combate ao Aedes aegypti

T7: Multi-agent modeling and simulation of an Aedes aegypti mosquito population

T8: DengueME: um framework de software para modelagem da dengue e seu vetor

T9: 2+ Dengue

T10: Traffic simulation using agent-based models

T11: Agent-based simulation framework for mixed traffic of cars, pedestrians and trams

2.5.4 Comparação

Observando a tabela comparativa da seção 2.5.3, verifica-se que os trabalhos T1, T2, T3, T4 e T5 contêm a característica de trabalhar com situações de alguma forma (observando as colunas "Plataforma de Simulação e Situação", "Situações Explícitas" e "Análise Visual de Situações". Dessa forma, as comparações seguintes apresentarão maiores detalhes sobre estes trabalhos.

O trabalho T1 apresenta uma plataforma que possibilita a simulação em razão da ocorrência de situações, de uma forma não explícita. Há uma forma de analisar a ocorrência de situações através do visual, no entanto, por não trabalhar com situações de forma explícita, a visualização é parcial. Além disso, a ferramenta possui um propósito específico: trabalhar com simulação de tarefas humanas. Cenários de simulação de tarefas humanas são auxiliados pelas situações, algo que justifica a implementação da plataforma de simulações e situações proposta. O desenvolvimento de modelos para declaração de situações das tarefas é interessante e útil, sendo uma forma de alto nível para descrever as situações e um ponto futuro a se inspirar no trabalho proposto.

T2 apresenta uma plataforma que faz uso de situações de forma explícita na simulação e possui análise visual das situações na ferramenta. No entanto, não é de propósito geral, tendo um propósito específico de trabalhar com simulação de redes. Este é mais um cenário justificado pelo uso de situações em simulações. A análise visual de situações da ferramenta é um ponto interessante a ser inspirado, mostrando situações de intrusão de sistemas na rede, algo que pode ser também útil a outros cenários e justificando a necessidade de uma ferramenta de uso geral.

T3 é uma plataforma que realizou testes em simulações para a detecção de situações de forma explícita e visual. O trabalho faz uso de um sistema simulador para testar situações que fazem o suporte a decisão de controle de navios. Apesar do trabalho ser relacionado, não é uma plataforma conjunta para simulação de situações. Assim, essa necessidade da experimentação justifica a elaboração do programa proposto, como uma plataforma única de simulação e situação. Apesar disso, a análise visual de situações se faz interessante para

inspiração e o novo cenário justifica a implementação de um programa para situações e simulações em propósitos gerais.

T4 também se apresenta como uma plataforma que foi simulada para a realização de detecção de situações de forma explícita e visual. Assim, há a detecção de situações em combate para auxílio a decisão. Assim como T3, os módulos de simulação e detecção de situações são separados, não sendo uma plataforma única para simulação e situação. Dessa forma, justifica a implementação da plataforma única proposta. Também, a análise visual de situações inspira a elaboração da plataforma proposta e o novo cenário justifica o uso de simulação e situação em uma forma generalizada.

T5 não se apresenta como uma plataforma tecnológica de simulação, realizando simulações em um ambiente médico simulado. A percepção de situação no procedimento médico é feita ao decorrer da simulação médica, com situações explícitas. Esse trabalho justifica o uso de situações em simulações, sendo uma forma mais próxima da realidade para a verificação de acontecimentos. Assim, a utilização da tecnologia com a simulação dos pacientes em uma plataforma poderia ser interessante para a verificação dos acontecimentos virtualmente, algo que justifica a implementação da plataforma proposta. Também, este é mais um cenário em que a plataforma proposta pode ser útil, uma vez que é de propósito geral.

Sendo assim, nenhum dos trabalhos relacionados apresenta a proposta de ser uma plataforma de simulação e situação com formas explícitas de trabalho com situações, análise visual de situações e que possua característica de simulação para diferentes cenários, sendo de propósito geral.

3 Situações e Simulações: Uma Pesquisa Exploratória

Este capítulo discute um estudo exploratório de diferentes abordagens para simulação baseado em um cenário envolvendo proliferação de mosquitos (*Aedes aegypti*). A pesquisa exploratória aborda cinco estratégias para simulação do cenário proposto: simulação orientada a objeto, simulação orientada a grafos, simulação orientada a regras e situações, simulação orientada a agentes inteligentes e simulação orientada a linguagens visuais. As abordagens foram escolhidas a partir de uma pesquisa entre as principais plataformas de simulação.

3.1 Simulações Situation-Aware

Como um novo conceito abordado nessa dissertação, as Simulações *Situation-Aware* (*SiSAs*) são simulações capazes de monitorar e reagir a determinados padrões de comportamento (Situações). Essas simulações demonstram visualmente a ocorrência de situações (Seção 2.1), incluindo o seu espaço e tempo, com a localidade (plano X, Y) e seu momento de ativação e desativação (visualização de um artefato representando a situação).

As *SiSAs* são interessantes para se realizar estudos de fenômenos complexos em diversos contextos, oferecendo suporte ao desenvolvedor na observação de situações ocorridas ao longo da simulação que possam vir a ser úteis, permitindo sua visualização no momento da ocorrência de ativação.

Uma *SiSA* fundamentalmente é formada por estruturas que contém um comportamento próprio (como agentes, por exemplo), eventos que indicam determinados acontecimentos na simulação e situações que ocorrem ao detectar padrões de eventos ou de estruturas.

Este tipo de simulação diferencia-se das outras simulações em sua característica de funcionamento, com uma programação de reação das entidades que compõem a simulação em uma forma similar aos acontecimentos na vida real. Ou seja, há a ocorrência de situações espaciais e temporais tal como ocorrem na vida real guiando a simulação.

3.2 Metodologia da Pesquisa

O primeiro passo para a pesquisa exploratória de simulação de situações é a busca por *softwares* / linguagens que possibilitem a implementação de simulações observando-se

a possibilidade de especificar o comportamento das entidades ou agentes de simulação, tal como uma *SiSA*. O ideal para uma ferramenta encontrada seria a possibilidade de análise por situações no resultado da simulação, no entanto, isso não foi encontrado.

Em seguida, as ferramentas são analisadas em função de seu código (aberto ou fechado), linguagem de programação da simulação (própria, visual, *Java*, entre outras), tipo de análise que a ferramenta proporciona (proporciona uma análise visual, quantitativa, em função do comportamento, entre outras...), facilidade de uso em relação a linguagem de programação (se é bem documentada, possui algum manual ou necessita de conhecimento prévio), utilização pela comunidade acadêmica (se é utilizada e como é utilizada) e qual a ideia principal de cada um dos *softwares* de simulação. As análises das ferramentas são importantes para verificar a possibilidade de integração. Sendo assim, o ideal seria uma ferramenta de código aberto que possua uma análise em razão do comportamento (situações), com uma boa documentação (sendo mais fácil de usar), uma utilização grande pela comunidade acadêmica (sendo uma ferramenta mais tradicional no campo acadêmico na realização de simulações) e com a possibilidade de se realizar diversos tipos de simulação (ferramenta genérica).

A partir da pesquisa é possível observar as ferramentas que possam cumprir os requisitos de uma *SiSA*, possibilitando a definição de um comportamento de situações ocorridas na realidade.

Escolhendo-se algumas ferramentas encontradas ao decorrer da pesquisa exploratória, uma análise das ferramentas pode ser realizada utilizando uma simulação do *Aedes aegypti* e, depois, analisando e comparando os resultados dessa simulação em razão das análises que ela possibilita, o desempenho da simulação e o tipo de programação da simulação.

A comparação e a análise são fundamentais para se observar ferramentas que efetivamente cumpram os requisitos de uma *SiSA* e possibilitem a análise de situações dos resultados da execução da simulação.

3.3 Ferramentas de Simulação

Uma pesquisa com os conceitos da *SiSA* obteve 21 linguagens de programação / *Softwares* a partir de strings de busca a respeito da *SiSA*, sendo eles: *AgentSheets* (REPENNING, 1993), *Altreva Adaptive Modeler* (PAWLEWSKI; DOSSOU; GOLINSKA, 2012), *AnyLogic* (YANG; LI; ZHAO, 2014), *AToM- tool for multi-paradigm modelling* (LARA; VANGHELUWE, 2002), *Breeve* (KLEIN, 2003), *Cougaar* (HELSINGER; THOME; WRIGHT, 2004), *DigiHive* (KORB; RANDALL; HENDTLASS, 2009), *Framsticks* (KOMOSIŃSKI; ULATOWSKI, 1999), *Groove* (GHAMARIAN et al., 2012), *Jade* (BELLIFEMINE et al., 2005), *Java* (GOSLING et al., 2014), *Mason* (LUKE et al., 2005), *Modelling4all*

Behavior Composer (KAHN et al., 2010), *Netlogo* (TISUE; WILENSKY, 2004), *Processing* (FRY; REAS, 2018), *RePast* (COLLIER, 2003), *Scene* (PEREIRA; COSTA; ALMEIDA, 2013), *StarLogo* (KLOPFER; BEGEL, 2003), *SugarScape* (AXELROD, 1997), *Swarm* (CHRIS, 1996), *TerraME* (CARNEIRO et al., 2004) e *UrbanSim* (WADDELL, 2002).

As linguagens de programação / *softwares* foram então analisados, apresentando a análise na próxima Seção.

3.4 Análise das Ferramentas de Simulação

As Tabelas 2 a 22 apresentam um levantamento de alguns dos principais *softwares* e linguagens para simulação e seus pontos em relação ao propósito principal, tipo de código (aberto / fechado), linguagem, tipo de análise que é possível realizar, facilidade de uso e a utilização pela comunidade acadêmica.

As Tabelas 2 a 6 apresentam as ideias principais dos *softwares* e como os *softwares* são utilizados nas simulações, isto é, se os *softwares* são de uso genérico em simulações ou uso específico além de suas particularidades.

Especificamente a linguagem de programação *Java* foi escolhida em razão da compatibilidade com a maioria dos *softwares* encontrados. A Tabela 2 apresenta alguns detalhes da linguagem em relação a simulações.

Tabela 2 – Linguagem de Programação - Ideia Principal

Linguagem	**Detalhes**
Java	O *Java* é uma linguagem de programação bastante utilizada. Por ser de uso geral, permite a criação de simulação em relação aos objetos.

A Tabela 3 apresenta os detalhes a respeito de *softwares* para simulação por grafos.

Tabela 3 – Simulação por Grafos - Ideia Principal

Software	**Detalhes**
AToM - tool for multi-paradigm modelling	O *Atom3* é uma ferramenta para se modelar e simular formalismos em grafos, transformando-os e modificando-os de um formalismo para outro.
Groove	O *Groove* é um *software* de propósito geral que utiliza a reescrita de grafos na aplicação de regras e exploração dos espaços de estados em simulações e verificações de sistema.

A Tabela 4 apresenta os detalhes a respeito de *softwares* para simulação por agentes.

Tabela 4 – Simulação por Agentes - Ideia Principal

Software	Detalhes
Breve	O *Breve* é um *software* que possui um ambiente 3D para se trabalhar com simulações com agentes utilizando-se ações descritas em linguagem *Python*.
Cougaar	O *Cougaar* é um *software* fechado mantido por uma empresa que faz uso de multiagentes e inteligência artificial para o planejamento militar em batalhas e questões de segurança. O acesso ao *software* é restrito.
Jade	*Jade (Java Agent DEvelopment Framework)* é um *framework* que ajuda no desenvolvimento de simulações utilizando-se a abstração de agentes.
Modelling4all	O *Modelling4All* é uma ferramenta online para permitir o desenvolvimento de modelos de agentes de forma rápida e simples.
Netlogo	*Netlogo* é uma ferramenta que possui a capacidade de modelar experimentos em multiagentes através de uma interface gráfica e linhas de comando.
RePast	*RePast* é um *framework* para facilitar a implementação de simulações utilizando agentes que tem a possibilidade de ser utilizado em diferentes linguagens de programação. Possui uma *IDE* pronta para ser utilizada com a visualização dos agentes simulados.
Swarm	*Swarm* é um *software* para modelagem de agentes utilizando-se a definição do comportamento de cada agente e visualizando-os em uma janela.

A Tabela 5 apresenta os detalhes a respeito de *softwares* para simulação visual.

Tabela 5 – Simulação Visual - Ideia Principal

Software	Detalhes
AgentSheets	O *AgentSheets* é um *software* utilizado para o desenvolvimento de aplicações, jogos e simulações utilizado por pessoas de diversas idades para programar de forma simples e visual.
Framsticks	O *Framsticks* é um ambiente tridimensional para se realizar simulações utilizando computação evolutiva e inteligência artificial em criaturas tridimensionais virtuais para física, biologia, medicina, neurociência, realidade virtual, robótica, etc.
Mason	*Mason* é uma biblioteca *Java* para ajudar na realização de simulações 2D e 3D.
Processing	*Processing* é uma linguagem de programação e *IDE* que possui uma utilização simples e possibilita a construção de artefatos visuais para artes, jogos, simulações, entre outros.
StarLogo	*Starlogo* é um *software* online para simulação e programação, utilizado prioritariamente no ensino de programação à crianças.

A Tabela 6 apresenta os detalhes a respeito de outros *softwares* para simulação.

Tabela 6 – *Outros Softwares* - Ideia Principal

Software	Detalhes
Altreva Adaptive Modeler	*Altreva* é um *software* que faz a simulação utilizando multi agentes sobre o mercado financeiro. Possui diversos componentes de visualização em gráficos e componentes que fazem o *download* de informações automaticamente do mercado.
Anylogic	*AnyLogic* é um *software* de simulação sobre os acontecimentos em ambientes (cidades e equipamentos) que faz uso de vários métodos de simulação (eventos discretos, agentes, e dinâmica de sistemas). Muito utilizado para o planejamento de ações.
DigiHive	O *DigiHive* é um ambiente para se realizar simulações em partículas e realizar a visualização do que ocorre em sistemas biológicos.
Scene	*Scene* é uma plataforma para o desenvolvimento de sistemas baseados em situação utilizando o *JBoss Drools* na especificação de regras e situações.
SugarScape	*Sugarscape* é um *software* de simulação celular para verificar o comportamento de sociedades simuladas.
TerraME	O *TerraME* permite analisar e realizar simulações espaciais utilizando *databases* geográficos *(GIS)*. O *software* possibilita o funcionamento a partir de outras linguagens de programação.
UrbanSim	O *UrbanSim* é um *software* pago, utilizado por grandes cidades para o desenvolvimento de simulações em seus meios de transporte: ruas, estradas, transportes públicos, . . . e em planejamento de construções.

As Tabelas 7 a 10 apresentam os códigos e linguagens utilizadas pelas ferramentas de simulação.

A coluna "código" representa a licença de implementação da ferramenta para que, por exemplo, verifique se possam ser feitas modificações que ampliem as funcionalidades da ferramenta.

A coluna "linguagem" apresenta o tipo de linguagem que a ferramenta utiliza em sua composição, isto é, a linguagem que foi utilizada para a implementação da ferramenta. Essa coluna é necessária para o entendimento da verificação de linguagens que são compatíveis entre si e verificação da possibilidade de implementação de melhorias na ferramenta.

A Tabela 7 apresenta a licença e informações da linguagem de programação a respeito de *softwares* para simulação por grafos.

Tabela 7 – Simulação por Grafos - Licença e Linguagem

Software	Código	Linguagem
AToM - tool for multi-paradigm modelling	Código Aberto	*Python*
Groove	*Software* gratuito	*Java*

A Tabela 8 apresenta a licença e informações da linguagem de programação a respeito de *softwares* para simulação por agentes.

Tabela 8 – Simulação por Agentes - Licença e Linguagem

Software	**Código**	**Linguagem**
Breve	Código Aberto	*Python*
Cougaar	Fechado e Pago	Não especificado (código fechado)
Jade	Gratuito e em código aberto	Java
Modelling4all	Gratuito e online	Fechado (online)
Netlogo	Gratuito e aberto	Diversas Linguagens
RePast	Gratuito e aberto	Diversas Linguagens
Swarm	Gratuito e aberto	*Objective-C* e *Java*

A Tabela 7 apresenta a licença e informações da linguagem de programação a respeito de *softwares* para simulação visual.

Tabela 9 – Simulação Visual - Licença e Linguagem

Software	**Código**	**Linguagem**
AgentSheets	Fechado e pago	Não especificado (código fechado)
Framsticks	Gratuito e em código aberto	*Java*
Mason	Gratuito e em código aberto	*Java*
Processing	Gratuito e aberto	*Processing*, *Java*, *Python*, entre outras
StarLogo	Online (fechado)	Fechado (online)

A Tabela 10 apresenta a licença e informações da linguagem de programação a respeito de outros tipos de *softwares* para simulação.

Tabela 10 – Outros *Softwares* - Licença e Linguagem

Software	**Código**	**Linguagem**
Altreva Adaptive Modeler	Versões gratuitas e pagas	Não especificado (código fechado)
Anylogic	Fechado e Pago	Não especificado (código fechado)
DigiHive	Gratuito para uso, código não disponibilizado, apenas ambiente	Não especificado (código fechado)
Scene	Biblioteca disponível para uso gratuito	*Java*
SugarScape	Gratuito, não disponível (fechado)	Fechado
TerraME	*Open-Source*	*LUA* c/ interface ao *TerraLib*
UrbanSim	Fechado e Pago	Não especificado (código fechado)

As Tabelas 11 a 14 apresentam o tipo de análise que a ferramenta de simulação proporciona ao programador. Por exemplo, se a ferramenta possibilita a visualização das

entidades ou agentes da simulação e como é essa visualização. As análises por visualização e por situações são relevantes para o trabalho proposto.

A Tabela 11 apresenta o tipo de análise proporcionado a respeito de *softwares* para simulação por grafos.

Tabela 11 – Simulação por Grafos - Tipos de Análise

Software	**Tipo de Análise**
AToM - tool for multi-paradigm modelling	Manipulação e exibição de modelos em grafos.
Groove	Análise visual (grafos) e quantitativa.

A Tabela 12 apresenta o tipo de análise proporcionado a respeito de *softwares* para simulação por agentes.

Tabela 12 – Simulação por Agentes - Tipos de Análise

Software	**Tipo de Análise**
Breve	Transformações em Ambiente 3D.
Cougaar	Análise gráfica, em mapas, etc.
Jade	Análise na própria programação do agente.
Modelling4all	Visualização 2D.
Netlogo	Visualização 2D e dados.
RePast	Estatística e visual (2D).
Swarm	Visualização 2D.

A Tabela 13 apresenta o tipo de análise proporcionado a respeito de *softwares* para simulação visual.

Tabela 13 – Simulação Visual - Tipos de Análise

Software	**Tipo de Análise**
AgentSheets	O *software* é utilizado para realizar programação visual, utilizando-se de blocos para montagem da programação. Dessa forma, a simulação é totalmente implementada, assim como a análise da mesma.
Framsticks	Análise visual (visualização 3D).
Mason	Visualização 2D e 3D.
Processing	Análise visual.
StarLogo	Visual (2D).

A Tabela 14 apresenta o tipo de análise proporcionado a respeito de outros tipos de *softwares* para simulação.

Tabela 14 – *Outros Softwares* - Tipos de Análise

Software	**Tipo de Análise**
Altreva Adaptive Modeler	Realiza análises visuais gráficas, planilhas e matemáticas.
Anylogic	Realiza visualização em três dimensões, gráficos e mapas (GIS).
DigiHive	Análise em visualização de partículas (visual).
Scene	Análise em situações e regras.
SugarScape	Visual (células).
TerraME	Visualização 2D (cores no mapa) e análise estatística.
UrbanSim	Visualização 3D e análise estatística.

As Tabelas 15 a 18 apresentam a facilidade de uso que o *software* proporciona ao programador em relação a utilização do ambiente para a programação da simulação. Por exemplo, se o *software* contém uma documentação online ou um ambiente pronto para instalar e usar.

A Tabela 15 apresenta a facilidade de uso proporcionada a respeito de *softwares* para simulação por grafos.

Tabela 15 – Simulação por Grafos - Facilidade de Uso

Software	**Facilidade de Uso**
AToM - tool for multi-paradigm modelling	Necessita da modelagem de formalismos em grafos para se realizar uma simulação.
Groove	Necessita da modelagem de formalismos em grafos para se realizar uma simulação.

A Tabela 16 apresenta a facilidade de uso proporcionada a respeito de *softwares* para simulação por agentes.

Tabela 16 – Simulação por Agentes - Facilidade de Uso

Software	Facilidade de Uso
Breve	Necessita do conhecimento da linguagem (*Python*) ou *script* próprio.
Cougaar	O *Software* é pronto e destinado para uma tarefa específica, cabendo a empresa o treinamento.
Jade	Utilização do *Java* para programação e uma documentação extensa.
Modelling4all	Possui manuais para interação online da ferramenta.
Netlogo	Dispõe de uma documentação online ensinando como utilizá-lo.
RePast	Apresenta manuais e documentação online ensinando como utilizá-lo.
Swarm	Dispõe de uma página de *Wiki*.

A Tabela 17 apresenta a facilidade de uso proporcionada a respeito de *softwares* para simulação visual.

Tabela 17 – Simulação Visual - Facilidade de Uso

Software	Facilidade de Uso
AgentSheets	Simples de utilizar, no entanto, é linguagem de blocos.
Framsticks	É necessário entender como fazer a programação do Framsticks através de um extenso manual.
Mason	Possui um vasto manual, ensinando como usar a ferramenta.
Processing	Disponibiliza documentação online e *IDE* prontas para uso.
StarLogo	Apresenta instruções online de uso.

A Tabela 18 apresenta a facilidade de uso proporcionada a respeito de outros tipos de *softwares* para simulação.

Tabela 18 – *Outros Softwares* - Facilidade de Uso

Software	Facilidade de Uso
Altreva Adaptive Modeler	Exige um conhecimento da área de finanças mas contém uma interface simples, de arrastar e soltar componentes.
Anylogic	Engloba diversas bibliotecas prontas que facilitam o desenvolvimento da simulação. Mapas e objetos 3D podem ser usados.
DigiHive	O *Software* é um ambiente para simular sistemas biológicos utilizando uma especificação em *xml*.
Scene	Instrui um passo-a-passo simples no *github* para instalação.
SugarScape	Disponibiliza um manual para uso.
TerraME	Necessita de modelos iniciais (*database*) e programação para se realizar a simulação.
UrbanSim	Não há programação, há o uso de *datasets* prontos e simulação na nuvem (cenários prontos para uso pela empresa).

As Tabelas 19 a 22 apresentam a maturidade do *software* na comunidade acadêmica, por exemplo, se o *software* é bastante utilizado atualmente ou apresenta alguma descontinuidade no processo de atualizações.

A Tabela 19 apresenta a maturidade na comunidade acadêmica a respeito de *softwares* para simulação por grafos.

Tabela 19 – Simulação por Grafos - Utilização

Software	**Utilização pela comunidade acadêmica**
AToM - tool for multi-paradigm modelling	Bastante utilizado uma vez que o *software* é aberto e mantido por uma universidade.
Groove	Utilizado pela comunidade acadêmica para experimentos com simulações e verificação de sistemas.

A Tabela 20 apresenta a maturidade na comunidade acadêmica a respeito de *softwares* para simulação por agentes.

Tabela 20 – Simulação por Agentes - Utilização

Software	**Utilização pela comunidade acadêmica**
Breve	Não utilizado mais, o *software* encerrou o suporte em 2015.
Cougaar	Não utilizado, *software* fechado para uso militar.
Jade	Bastante utilizado pela comunidade acadêmica em experimentos de comunicações e redes. Mantido pela telecom italia.
Modelling4all	Utilizado para criação de modelos de agentes de forma simples e online.
Netlogo	Bastante utilizado pela comunidade acadêmica em experimentos.
RePast	Bastante utilizado em diversos experimentos de propósitos gerais pela comunidade acadêmica.
Swarm	Pouco utilizado pela comunidade (parece que o projeto foi abandonado).

A Tabela 21 apresenta a maturidade na comunidade acadêmica a respeito de *softwares* para simulação visual.

Tabela 21 – Simulação Visual - Utilização

Software	Utilização pela comunidade acadêmica
AgentSheets	Pouco utilizado, sendo mais utilizado para a construção de jogos.
Framsticks	Bastante utilizado pela comunidade acadêmica em experimentos de química, física e biologia.
Mason	Bastante utilizado pela comunidade acadêmica em experimentos de vida artificial e física.
Processing	Utilizado pela comunidade acadêmica para diversos experimentos e pela comunidade em geral para artes gráficas.
StarLogo	Pouco utilizado, apenas para ensino de programação a crianças.

A Tabela 22 apresenta a maturidade na comunidade acadêmica a respeito de outros tipos de *softwares* para simulação.

Tabela 22 – *Outros Softwares* - Utilização

Software	Utilização pela comunidade acadêmica
Altreva Adaptive Modeler	Pouco utilizado, sendo mais utilizado por profissionais que buscam fazer simulações com o interesse no mercado de ações.
Anylogic	Pouco utilizado pela restrição de acesso.
DigiHive	Pouco utilizado pela comunidade.
Scene	Utilizado pela comunidade para a construção de sistemas baseados em situação.
SugarScape	Utilizado pela comunidade para o desenvolvimento de experimentos em sociedade artificial.
TerraME	Bastante utilizado (*INPE* mantém o software).
UrbanSim	Pouco utilizado pela restrição de acesso.

É desejável que os *softwares* / linguagens sejam de código aberto, com uma linguagem compatível entre si, dessa forma há a possibilidade de se realizar uma implementação unindo dois ou mais *softwares* com o objetivo de melhorar a sua funcionalidade fazendo uma junção entre os melhores pontos de cada um deles.

Também, é desejável que sejam complementares na análise da simulação, assim um *software* poderá se utilizar da análise de outro *software* para uma complementação. Por exemplo, se um *software* possibilita a análise por visualização de simulações e outro *software* possibilita uma análise estatística da simulação, podem ser complementares.

A fácil utilização é um ponto interessante pois facilita o trabalho do desenvolvedor na introdução de um novo ambiente para simulação, melhorando inclusive a aderência do *software* pela comunidade acadêmica e desenvolvedores. Ter uma maior utilização pela comunidade acadêmica também faz com que o *software* seja reconhecido e tenha mais trabalhos utilizando-o.

Dessa forma, foram escolhidos cinco *softwares* / linguagens em uma pesquisa exploratória com o uso de cenário do *Aedes aegypti*. Para cada um dos *softwares* e linguagens de programação, foram observados possíveis pontos de melhoria para integrá-los em uma única plataforma de análise de situações em simulações. Como resultado dessa observação foram obtidos o *Java*, o *Groove*, o *Scene* e o *RePast* como os possíveis *softwares* a serem integrados.

A primeira escolha foi o *Java*, pela linguagem de programação amplamente utilizada com uma proposta de uso geral. A segunda escolha foi o *Groove* em razão de ser um *software* livre e com a possibilidade de se modelar simulações visualmente, utilizando grafos. A terceira escolha foi o *Scene*, com funções de análise e reação a contexto de acordo com situações na simulação. A quarta escolha foi o *RePast* (*ReLogo*) por utilizar uma linguagem de uso simples para composição de simulações utilizando agentes. Em quinto lugar o *Processing* pela capacidade visual de reprodução da simulação e diversas bibliotecas que facilitam a programação.

3.5 Cenário do *Aedes aegypti*

O *Aedes aegypti* é um mosquito presente principalmente no ambiente urbano de países com clima tropical e subtropical e é o principal vetor de diversas doenças (LOK; KIAT; KOH, 1977). A dinâmica do *Aedes aegypti* é complexa e tem motivado pesquisadores a realizar diversos estudos nas comunidades humanas (MORATO et al., 2005; VASCONCELOS, 2015).

As doenças arbovirais estão emergindo e ressurgindo em diferentes partes do mundo, evoluem rapidamente, tendo genótipos associados com aumento da virulência. A incidência de dengue, por exemplo aumentou 30 vezes com o aumento da expansão geográfica para novos países e, na presente década, passou de ambiente urbano para rural. Estima-se que 50 milhões de casos de dengue ocorrem anualmente e aproximadamente 2,5 bilhões de pessoas vivem em países endêmicos para dengue. O fardo da doença é alto, com mais de 500 milhões de casos por ano e isso requer ação imediata (SIVAGNANAME; GUNASEKARAN et al., 2012) (JOHNSON; RITCHIE; FONSECA, 2017).

O controle vetorial tem sido considerado um importante aliado na prevenção e controle das infecções – dengue, chicungunha, zika, febre amarela. Para que tal ação obtenha êxito, deve-se considerar que os vetores destas doenças reproduzem-se em uma variedade de habitats e preferencialmente em reservatórios de águas limpas. Embora essas espécies possam adaptar-se a outros tipos de criadouros, como bromélias e esgotos a céu aberto (BESERRA et al., 2009), são altamente resilientes, uma vez que são capazes de mudar os seus habitats de reprodução com muita frequência (SIVAGNANAME; GUNASEKARAN et al., 2012).

Como Sivagnaname et al (SIVAGNANAME; GUNASEKARAN et al., 2012) sugerem, reduzir a densidade dos vetores da dengue para níveis baixos é a única medida atualmente disponível para prevenir a transmissão da dengue, sem, no entanto, resultar num baixo nível de transmissão da doença. Um único vetor pode transmitir a doença a muitas pessoas pelo seu comportamento diurno, antropofágico, interrompido e múltiplo. Da mesma forma em que se dá com as demais doenças infecciosas transmitidas pelas espécies *Aedes aegypti* e *Aedes albopictus*, um único vírus pode ser suficiente para produzir uma infecção no hospedeiro humano.

Para a Organização Mundial da Saúde (OMS) há uma necessidade urgente de desenvolver ferramentas de controle de vetores para o controle sustentado de populações de *Aedes aegypti* e *Aedes albopictus*, principalmente em comunidades endêmicas. Algumas novas e promissoras ferramentas de controle de vetores de dengue são objeto de pesquisa e atualmente estão sendo testadas em campo para seu uso em intervenções de saúde pública, como materiais tratados com insecticida (ITMs) que consistem em redes insecticidas de longa duração, cortinas e tapeçarias; e armadilhas de oviposição letais ("Ovitraps") que recolhem os ovos colocados pelos mosquitos que se desenvolvem em larva, pupa e mosquitos adultos (World Health Organization, 2017) (SIVAGNANAME; GUNASEKARAN et al., 2012).

A utilização de tecnologias inteligentes aliadas a instrumentos eficazes de vigilância entomológica é necessária para a prevenção eficaz das doenças infecciosas transmitidas pelo *Aedes aegypti* e *Aedes albopictus* (JOHNSON; RITCHIE; FONSECA, 2017). Existe uma necessidade urgente de abordagens novas e inovadoras para um controle sustentável dos vetores (SIVAGNANAME; GUNASEKARAN et al., 2012).

Com o objetivo de desenvolver novas intervenções destinadas a reduzir tanto os mosquitos adultos do sexo feminino e seus futuros descendentes, os pesquisadores desenvolveram armadilhas de oviposição letais, que atraem e matam as fêmeas que realizam a oviposição no local, assim como seus descendentes que ficam presos na armadilha (JOHNSON; RITCHIE; FONSECA, 2017).

Um exemplo moderno de uma armadilha letal para o controle de mosquitos pode ser creditado a Lok et al. (LOK; KIAT; KOH, 1977). Consistiu em um recipiente cilíndrico preto, cheio de água, com um dispositivo de flutuação composto por um fio de malha e duas pás de madeira. Embora as fêmeas ovipositadas não tenham sido mortas, as larvas desenvolvidas na água sob a malha de arame e adultos emergentes ficaram presas (JOHNSON; RITCHIE; FONSECA, 2017). Essas armadilhas são conhecidas como *ovitraps* e necessitam de visitas periódicas dos agentes de saúde para a contagem manual dos vetores. A visita periódica ajuda na detecção de localidades com maior concentração dos mosquitos para realizar ações efetivas e centralizadas no combate às doenças transmitidas pelos vetores.

A simulação proposta na dissertação utiliza a armadilha como uma forma de verificar a existência de mosquitos em uma determinada região, tendo como objetivo a validação na distribuição da armadilha em um determinado espaço geográfico. Observa-se que, por ser uma simulação, a armadilha não está fisicamente presente mas seu funcionamento é simulado de forma que é possível colocar no ponto exato de uma localização e verificar o comportamento da armadilha em relação aos mosquitos da região.

Utilizando-se o panorama de uma determinada região (um bairro ou uma cidade, por exemplo), o cenário da simulação é formado por casas, mosquitos, ovos, agentes de combate ao vetor e armadilhas.

As casas representam os locais em que os mosquitos podem fazer a oviposição e podem possuir focos (criadouros do mosquito) ou armadilhas. No caso do mosquito fazer a oviposição em focos, há a evolução dos ovos para novos mosquitos após 20 dias (MAIMUSA et al., 2016). De outra forma, a oviposição em armadilhas faz com que o mosquito seja morto e os ovos não evoluam para novos mosquitos (World Health Organization, 2017) (SIVAGNANAME; GUNASEKARAN et al., 2012). O agente de saúde, ao fazer uma verificação dois dias após a oviposição do mosquito, consegue observar que há ovos e que é necessário um controle vetorial do mosquito na região em que a armadilha está presente (DZUL-MANZANILLA et al., 2016). O controle consiste em fumacê e visita domiciliar, que faz com que ovos, mosquitos e focos presentes na região sejam exterminados (DZUL-MANZANILLA et al., 2016).

Os mosquitos sobreviventes permanecem fazendo a oviposição por até 37 dias (MAIMUSA et al., 2016). Durante a sua vida, os mosquitos são livres para sobrevoar e fazer a oviposição em até 100 metros do ponto em que nasceu (DZUL-MANZANILLA et al., 2016). Fenômenos climáticos também estão no cenário, de forma que os focos combatidos voltem após uma determinada chuva, por exemplo.

Tendo-se este cenário em vista, esta dissertação propõe a implementação da simulação do *Aedes aegypti* utilizando-se cinco abordagens diferentes: simulação orientada a objetos (*Java*), orientada a grafos (*Groove*), orientada a regras e situações (*Scene*), orientada a agentes inteligentes (*ReLogo*) e orientada a interfaces gráficas (*Processing*). As diferentes abordagens foram verificadas e validadas de acordo com informações científicas a respeito do comportamento do mosquito, simulando a dinâmica do mosquito nos ambientes urbanos e tendo a importância de ser uma ferramenta para suporte a decisão uma vez que pode ser utilizada para auxiliar profissionais de saúde a identificar locais com maiores incidências do *Aedes aegypti* (BALDI et al., 2017).

Como aspecto de espaço da simulação, foi utilizado o campus Goiabeiras da Universidade Federal do Espírito Santo. O campus possui 157 hectares de área, no qual 104 hectares são de áreas com construções. Fazendo uso do serviço *OpenStreetMaps*, foi realizada a extração de dados relacionados a todas as localizações dos edifícios no campus

para os programas em questão (OPENSTREETMAP, 2017).

Também, foram utilizados dados públicos a respeito do *Aedes aegypti* sobre as cidades de Vitória e Rio de Janeiro para a obtenção de uma estimativa da quantidade de mosquitos e armadilhas distribuídas nas cidades. Foi observado que em Vitória 1410 armadilhas foram distribuídas em 80 bairros em 2011 (PICCIN, 2013) (uma média de 17 armadilhas por bairro). Considerando o campus universitário um bairro, foi estabelecido o uso de 17 armadilhas espalhadas no campus para a simulação. Em relação aos mosquitos, utilizou-se os dados de um estudo no Rio de Janeiro (FREITAS; EIRAS; OLIVEIRA, 2008) em que há uma população de 10 mosquitos fêmea *Aedes aegypti* por hectare. Uma vez que o campus possui 104 hectares de área construída, foram considerados 1040 mosquitos como população inicial.

3.6 Simulação Orientada a Objetos

A primeira implementação realizada foi feita em *Java*, uma linguagem de alto nível, orientada a objeto e de propósito geral. A implementação em *Java* é realizada utilizando a manipulação de objetos. Objetos são formados por atributos que descrevem suas propriedades de acordo com o domínio no mundo real. Os atributos, então, são manipulados por condicionais *if-then-else* para a implementação do comportamento da simulação.

```
public class House {
    private List<House> neighbors = new ArrayList<House>();
    private List<Mosquito> mosquitos = new ArrayList<Mosquito>();
    private List<Eggs> eggs = new ArrayList<Eggs>();
    private List<Agents> agents= new ArrayList<Agents>();
    private boolean focus = false;
    private boolean trap = false;
    private boolean activefocus = false;
    private String name = "";
```

Figura 10 – Objeto “House”.

A implementação do cenário do *Aedes aegypti* é formada por listas de objetos. Há uma lista de objetos “*House*” (Figura 10), que são as casas, conectadas umas com as outras, representando a vizinhança na qual o mosquito pode migrar. O objeto “*House*” também contém referências às listas de objetos “*Mosquito*”, “*Eggs*” e “*Agent*”, indicando que possui mosquitos, ovos ou agentes nas casas respectivamente. Cada objeto possui também atributos para representar o estado do objeto. “*House*” possui booleanos que indicam se a casa é um foco, se o foco está ativo e se contém armadilhas. “*House*” também possui uma *String* para identificar o nome da casa em questão.

```
public class Eggs {
   private int days = 0;

    public int getDays() {
        return days;
    }

    public void setDays(int days) {
        this.days = days;
    }

}
```

Figura 11 – Objeto “Eggs”.

O objeto “*Eggs*” representa os ovos do mosquito e define um inteiro para controle de dias de vida dos ovos (Figura 11).

```
public class Mosquito {
   private int days = 0;
   private boolean control = false;
   private House born;
```

Figura 12 – Objeto “Mosquito”.

O objeto “*Mosquito*” define um inteiro para controle de dias de vida, ou seja,quantos dias o objeto em questão possui em razão do tempo de simulação. Também, há um booleano de controle para controlar o ato de voar / botar ovos, fazendo com que a simulação reconheça o objeto e não faça a ação de voar mais de uma vez ou botar mais ovos que o necessário em cada uma das “rodadas” ou dias de simulação. O mosquito também possui uma área de vôo previamente determinada em razão de suas capacidades, assim, há um ponteiro para a casa no qual o mosquito nasceu (Figura 12).

```
public class Agents {

    private int control = 3;

    public int getControl() {
        return control;
    }

    public void setControl(int control) {
        this.control = control;
    }

}
```

Figura 13 – Objeto “Agents”.

O objeto “*Agents*” representa os agentes de saúde de combate ao mosquito / ovos na simulação. Ele possui um ponteiro que percorre as casas vizinhas representando a “visita” do agente (Figura 13).

“*Scenary*” é o objeto que contém todas as casas, mosquitos, ovos e agentes (Anexo A) e define os métodos para executar a simulação.

A simulação em *Java* funciona a partir de *loops*, nos quais o comportamento de cada objeto da simulação é baseado em condicionais *if-then-else*. Cada *loop* da simulação

representa uma passagem de dias (ou rodadas) no qual o cenário de simulações sofreu modificações. A seguir serão apresentadas as ações que cada objeto exerce na simulação.

No método "fly" (Anexo A), encontra-se o código que define o comportamento de vôo do *Aedes aegypti* na simulação implementada em Java. Inicialmente, o código percorre a lista de casas verificando a lista de mosquitos de cada uma e, consequentemente, mosquito por mosquito. Uma vez encontrado o mosquito, o *booleano* de controle fornece a informação se o mosquito já efetuou o vôo através do método "*getControl()"*. Se o mosquito em questão não efetuou o vôo (*false*), a lista de casas vizinhas é acessada modificando o booleano com o método "*setControl(true)*". A partir da modificação, há a escolha de uma das casas vizinhas, possíveis do mosquito voar (método "*getNeighbors()*"), de forma aleatória (método "*randomize*") e atribuindo o objeto do mosquito à nova casa (método "*add(mosquito)*") e removendo objeto do mosquito da casa anterior (método "*remove(mosquito)*".

O comportamento de botar ovos (método "LayEggs" - Anexo A) faz com que a simulação percorra a lista de casas e, dentro da lista verifique a lista de mosquitos. Para cada mosquito encontrado na lista de mosquitos, o programa verifica a casa em que está ("*inHouse*"): se é uma casa com foco ("*isFocus()*") ou armadilha ("*isTrap()*"). Caso a a casa seja um foco, o método "*addEggs()*" adiciona objetos de ovos na casa. Caso seja uma armadilha, o método "*newAgent()*" cria agentes na casa.

Já no comportamento de ovos nascendo (método "hatchEggs" - Anexo A), o programa percorre a lista de casas e, em cada casa, percorre a lista de ovos. A partir de cada conjunto de ovos da lista de ovos, verifica se a idade dos conjuntos (método "*getDays()*") é de 20 dias. Caso seja, são adicionados 10 mosquitos por conjunto (método "*addMosquitoToHouse(house)*").

A função "*clearDay*" (Anexo A) é a responsável pela passagem de um dia da simulação. Ela percorre a lista de casas e em cada casa verifica as listas de mosquitos e de ovos. Também, o clima é simulado de forma que a cada resto 0 de 15 (a cada 15 dias), os focos inativos voltam a serem ativos (método "*setActivefocus(true)*", simulando a chuva. Para a lista de mosquitos, a função faz o incremento dos dias dos mosquitos (utilizando o método ''*setDays(dayMosquito)*") e modifica o *booleano* de controle (através do método "*setControl(false)*"). Caso os dias de vida do mosquito sejam de 38 dias, o objeto do mosquito é retirado da simulação (método "*remove(mosquito)*"). Para a lista de ovos, a função incrementa os dias que o objeto de ovos está presente na simulação através do método "*setDays(Eggs)*".

A função de comportamento do agente (método "Agent" - Anexo A) verifica as listas de agentes nas casas. A partir disso, cada agente presente na simulação (criado pela armadilha anteriormente) possui uma estrutura de controle em dias ("*control*"). Essa estrutura é incrementada a cada iteração na função do objeto agente (a cada

dia), verificando se o agente já está no terceiro dia da simulação. Caso esteja, o agente inicia a sua ação: faz a remoção de qualquer outro agente que esteja sendo chamado (método "*remove(verifyingAgents)*"), remove todos os elementos da lista de ovos da casa em que está (método "*clear()*"). Em seguida, percorre outras casas vizinhas, removendo os ovos e mosquitos (método "*clear()*") além de modificar o booleano de foco (método "*setActivefocus(false)*") para indicar que a casa não é mais um foco do mosquito.

A visualização da simulação em *Java* é realizada utilizando o retorno de mensagens ao programador a respeito das mudanças de entidades na simulação. Dessa forma, há a contagem de mosquitos e ovos ao decorrer da simulação.

3.7 Simulação Orientada a Grafos

Uma segunda implementação foi realizada na ferramenta *Groove* (*GRaph based Object-Oriented VErification*). O *Groove* utiliza grafos para representar os estados de um sistema e aplica regras de transformações em grafos (reescrita) para modelar sistemas dinâmicos (GHAMARIAN et al., 2012), como apresentado no referencial teórico (Seção 2.2.1) .

Os elementos do cenário foram modelados no *Groove* utilizando nós identificados, isso é, nós que possuem um tipo em sua estrutura. Por exemplo, um nó identificado como "casa" representa uma casa.

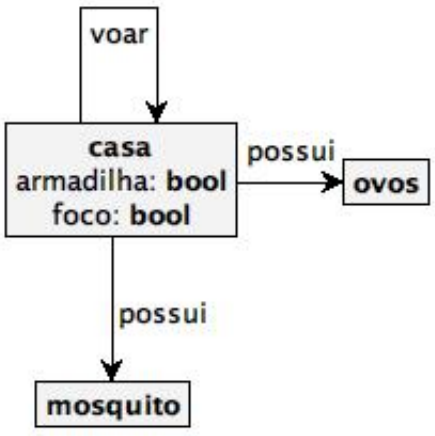

Figura 14 – Tipo "*Casa*"

Na Figura 14 pode-se observar o grafo que representa o tipo "*casa*". O nó "*casa*" define dois atributos para identificação se a casa contém armadilha ou foco. Além disso, o nó "*casa*' define as arestas "*voar*" e "*possui*" indicando que obrigatoriamente deve voar para um outro nó "*casa*" e pode possuir "*ovos*" ou "*mosquito*".

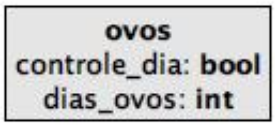

Figura 15 – Tipo "*Ovos*"

O tipo "*ovos*" (Figura 15) define um nó com dois atributos: um atributo para controle para passagem de dias ("*controle_dia*"') e um atributo de vida "*dias_ovos*".

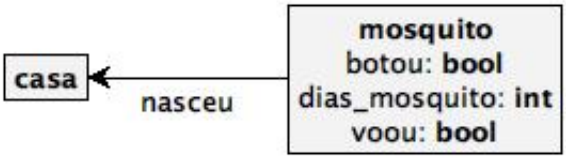

Figura 16 – Tipo "*Mosquito*"

O tipo "*Mosquito*" (Figura 16) define um nó "*mosquito*" com os atributos de controle da simulação "*botou*" e "*voou*" que controlam se o mosquito já botou ou voou naquele dia respectivamente. O nó "*mosquito*" também define um atributo de controle inteiro "*dias_mosquito*" para verificação de dias de vida do mosquito. Além disso, cada nó "*mosquito*" deve obrigatoriamente conter uma aresta "*nasceu*" para uma casa.

Figura 17 – Tipo "*Agente*"

O tipo "*agente*" (Figura 17) contém um nó "*agente*" com um contador de visitas para contar as casas visitadas e obrigatoriamente este nó deve estar conectado a um nó "*casa*" com o rótulo "*visita*".

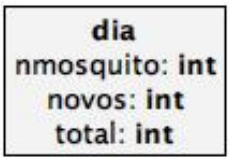

Figura 18 – Tipo "*Dia*"

O tipo "*dia*" (18) é utilizado para o controle de dia da simulação e visualização dos dados da simulação. O nó "*dia*" é formado por um contador de número de mosquitos

e ovos da simulação ("*nmosquito*" e "*novos*" respectivamente) e por um contador do total de dias passado pela simulação ("*total*").

Com os tipos definidos, é realizada a programação das regras de transformação dos grafos em função de grafos de reescrita. A seguir serão apresentados os grafos de reescrita que apresentam o comportamento da simulação.

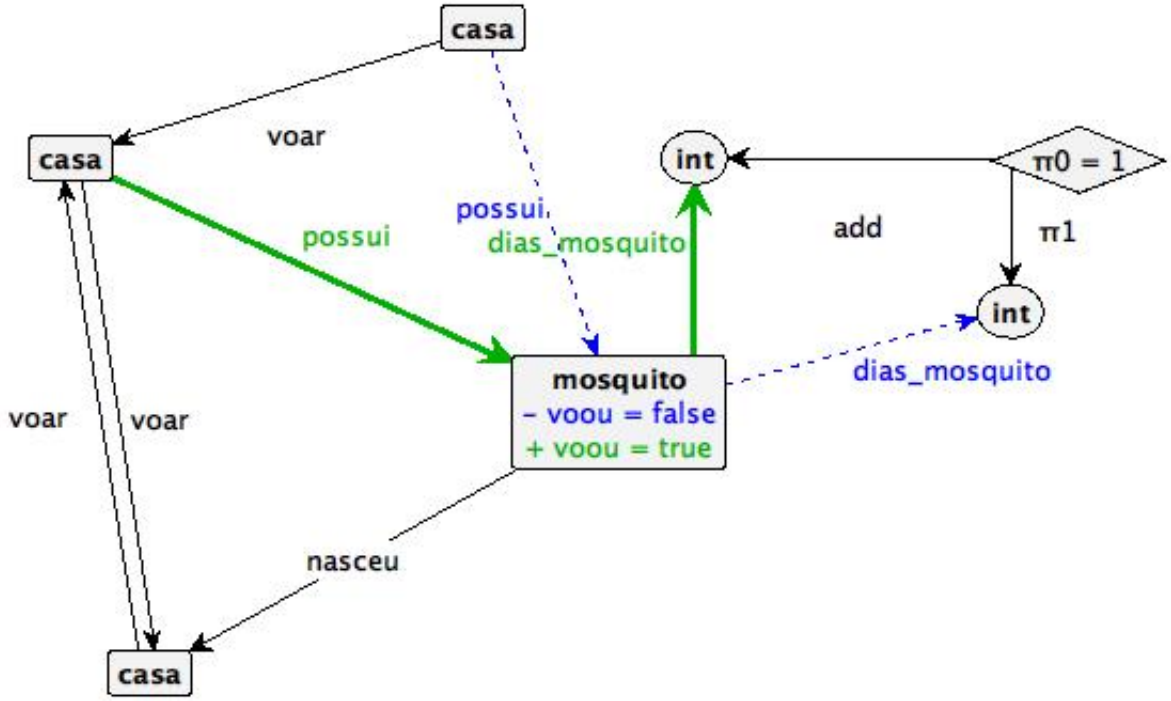

Figura 19 – Comportamento de Vôo do Mosquito

Para o vôo do mosquito, a reescrita (Figura 19) é feita de forma que o booleano "*voou*" do nó mosquito deixa de ser falso para ser verdadeiro, efetuando o controle para que cada mosquito voe uma vez por dia. Em seguida, o inteiro "*dias_mosquito*" é incrementado, incrementando em um dia a vida do mosquito. Uma nova casa é escolhida uma vez que a aresta "*possui*" deixa o nó "*casa*" anterior para escolher um novo nó "*casa*".

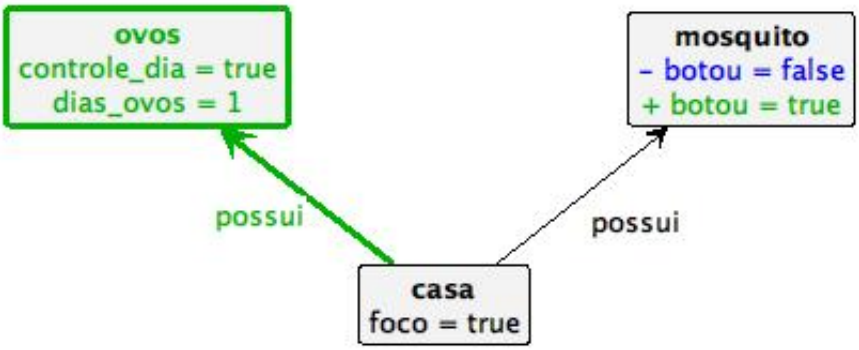

Figura 20 – Comportamento de Botar Ovos em Foco

Para o comportamento de botar ovos em um foco, a reescrita (Figura 20) faz com que um novo nó "*ovos*" seja criado e o booleano"*botou*" do mosquito deixa de ser falso para ser verdadeiro.

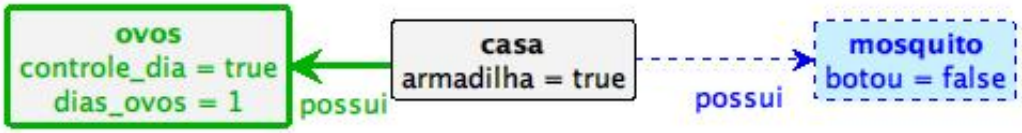

Figura 21 – Comportamento de Botar Ovos em Armadilha

Já o comportamento de botar ovos em armadilha (Figura 21) ocorre caso o mosquito bote em uma armadilha e a reescrita faz com que o nó "*ovos*" seja criado e o nó "*mosquito*" deixe de existir.

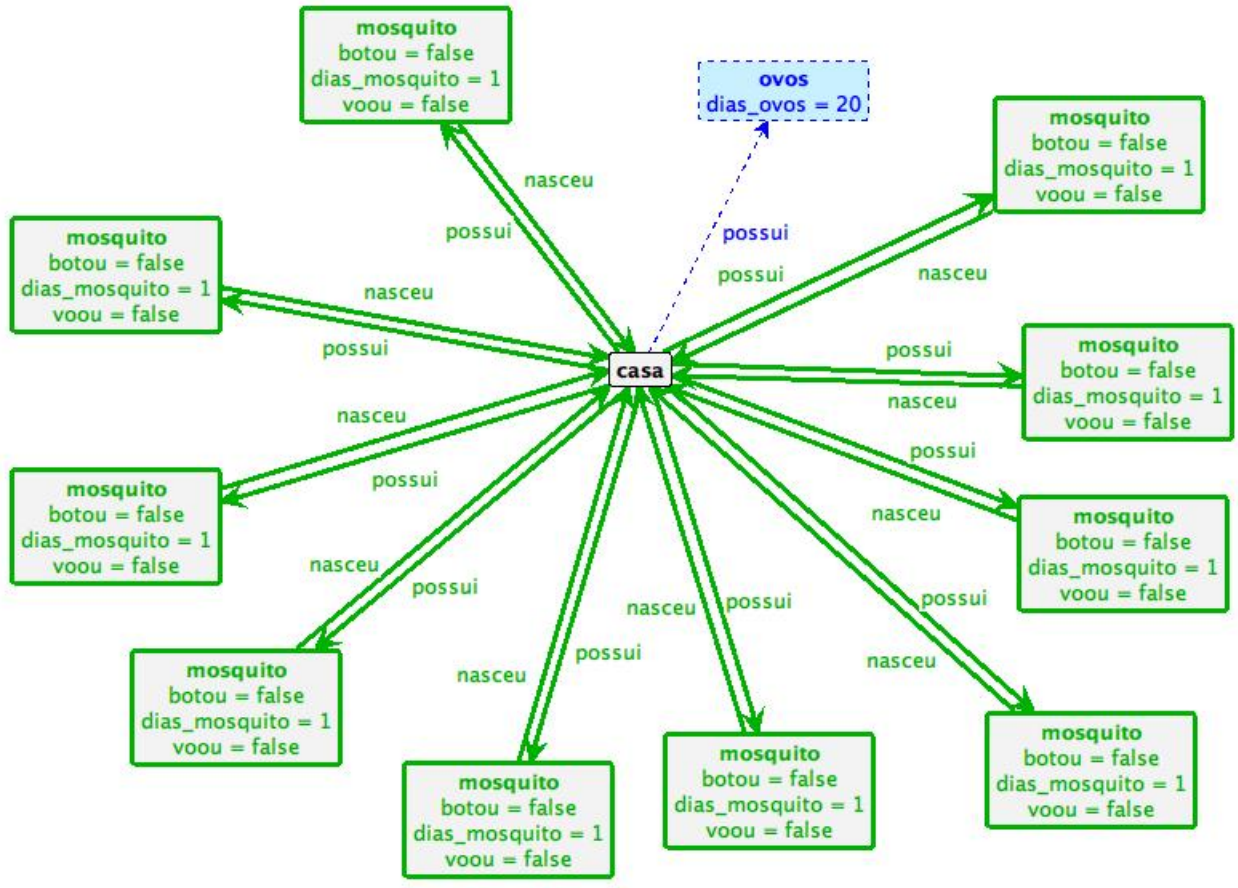

Figura 22 – Comportamento de Eclosão dos Ovos

Para o comportamento do conjunto de ovos nascendo, ou seja, ovos eclodindo, ocorre a reescrita (Figura 22) uma vez que o nó "*ovos*" atinge o tempo de vida de 20 dias, sendo este indicado pelo inteiro "*dias_ovos*". Com a eclosão, 10 novos nós "*mosquito*" são criados.

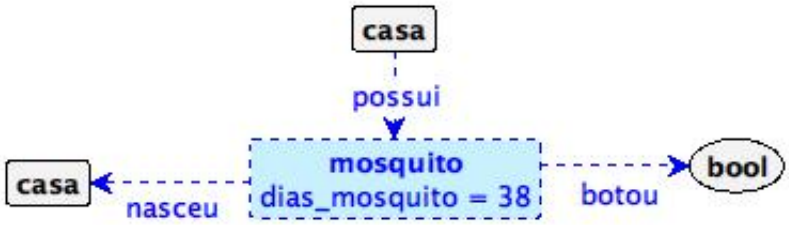

Figura 23 – Comportamento de Vida do Mosquito

O mosquito possui um tempo de vida e, dessa forma, os nós "*mosquito*" ao chegarem ao dia 38 (através do controle do "*dias_mosquito*" são retirados da simulação com a reescrita (Figura 23), simulando sua morte.

Figura 24 – Comportamento da Armadilha Inteligente

Para a criação dos agentes de saúde em uma armadilha inteligente, a reescrita (Figura 24) é realizada uma vez que o nó "*ovos*" com "*dias_ovos*" chega a 3 em uma casa com o booleano "*armadilha*" verdadeiro. Um nó "*agente*" com o controle "*visitas*" igual a 5 é criado, simulando a chamada ao agente.

Figura 25 – Comportamento de Combate aos Mosquitos

Figura 26 – Comportamento de Combate aos Ovos

O nó "*agente*" por sua vez, ao estar ligado a uma "*casa*" conectado ao nó "*mosquito*" (Figura 25) ou "*ovos*" (Figura 26), faz a retirada de qualquer um dos dois nós, simulando o combate dos mosquitos e ovos.

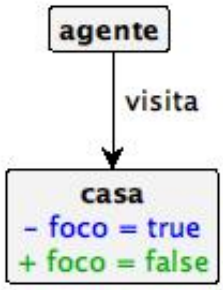

Figura 27 – Comportamento de Combate ao Foco

Também, o nó "*agente*" ao visitar um nó "*casa*" com o controle "*foco*" verdadeiro (Figura 27), passa a ser falso, simulando a desativação de focos em uma casa.

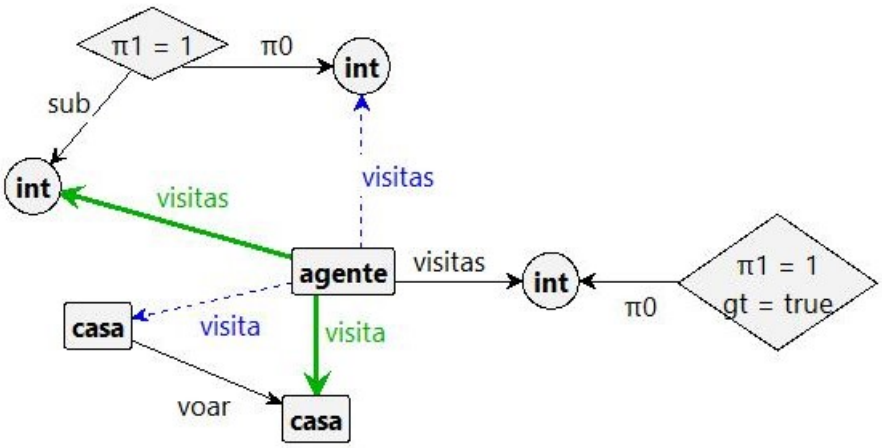

Figura 28 – Comportamento de Percorrer Casas pelo Agente

Finalmente, o agente visita e percorre as casas, decrementando as visitas nas casas (5 visitas iniciais) como visto na reescrita da Figura 28.

Figura 29 – Comportamento de Encerramento do Agente

Ao concluir a visita às casas, o nó "*agente*" é excluído da simulação, como na reescrita da Figura 29.

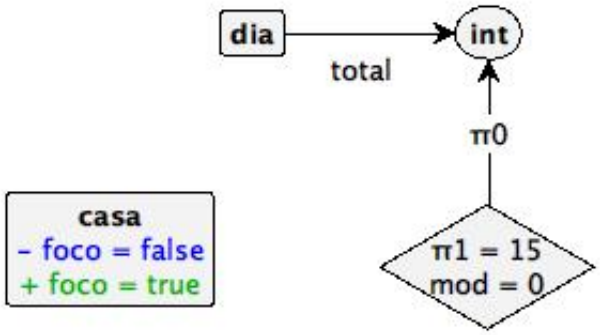

Figura 30 – Comportamento da Chuva

Para simular a chuva, quando o atributo "*total*" do nó "*dia*" possui um resto de divisão 15, os nós "*casa*" que possuem atributo de "*foco*" falso, viram verdadeiros (Figura 30.

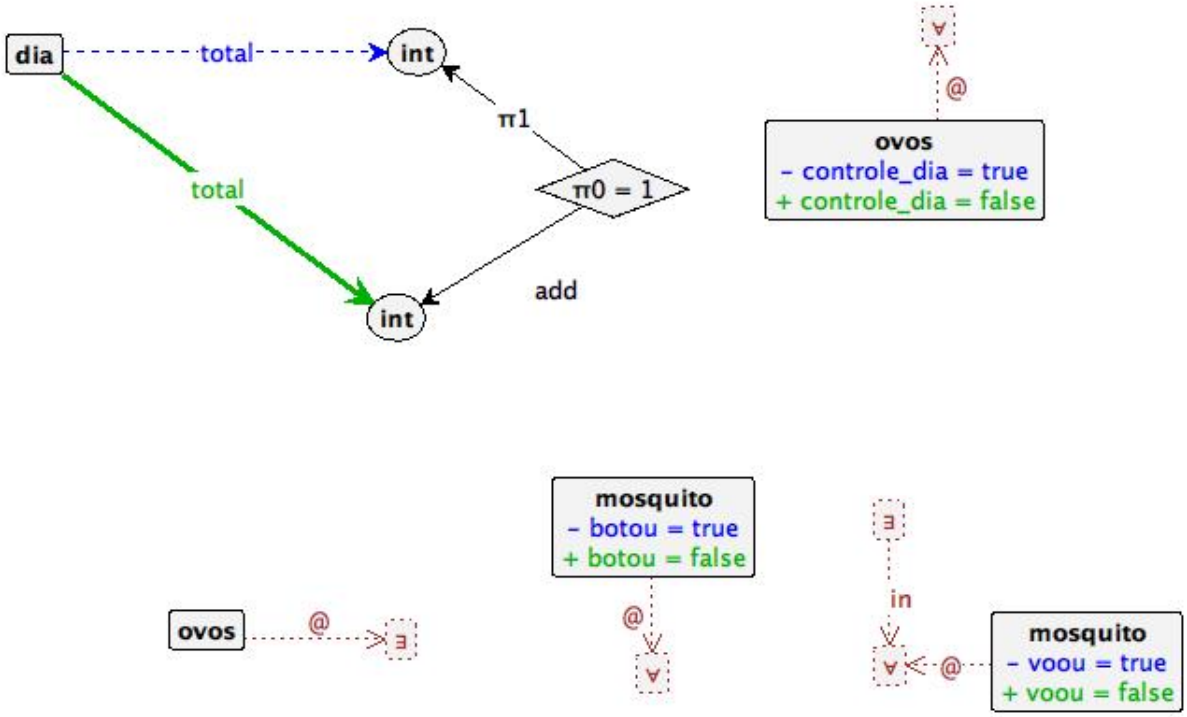

Figura 31 – Controle de Dias / Rodadas

O controle de dias da simulação e controle dos booleanos é feito em razão de rodadas ou dias de simulação, conforme as reescrita da Figura 31. Para cada dia passado, o número total de dias é incrementado. Os controles de dia dos ovos (para incrementar os dias dos ovos em cada rodada), o controle de vôo do mosquito (para voar a uma casa em cada dia) e o controle de botar do mosquito (para botar apenas uma vez) é atribuído a falso. As notações *"@"* ao símbolo "para todo" servem para indicar que essa reescrita pode ser feita uma única vez para todos os ovos e mosquitos.

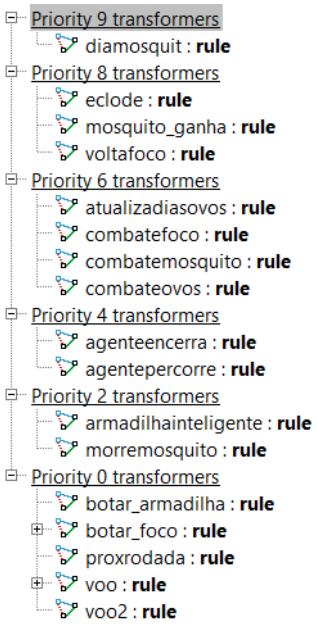

Figura 32 – Controle de Prioridades de Transformações

A Figura 32 exibe as prioridades das reescritas comportamentais anteriormente exibidas. A prioridade 9 (parte superior da Figura) possui as maiores prioridades de reescrita, estando as outras setadas como prioridades inferiores 8, 6, 4, 2 e 0 respectivamente. Uma reescrita de eclodir ovos ("*eclode*"), por exemplo, deve ter uma prioridade maior que a reescrita de passagem de rodada ("*proxrodada*") para que os ovos eclodam antes da passagem de dias, com uma ordem no processo de simulação.

A visualização da simulação em Groove é feita a partir da transformação do cenário inicial representado por "nós". Cada "nó" pode sofrer alterações, sendo excluído, modificado ou criado na representação de uma entidade da simulação (mosquitos, ovos, agentes).

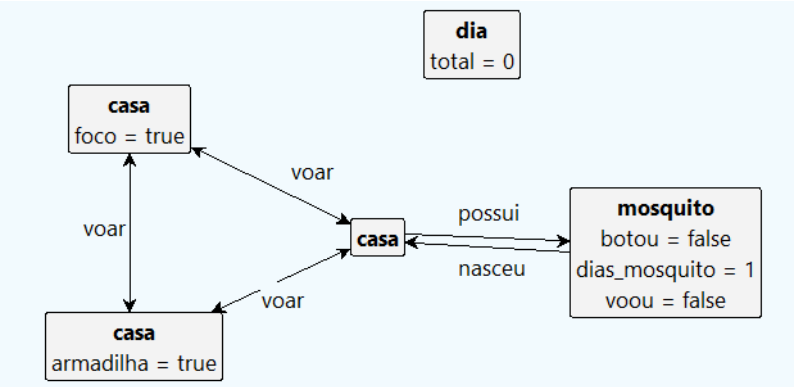

Figura 33 – Cenário Inicial de Exemplo- Groove

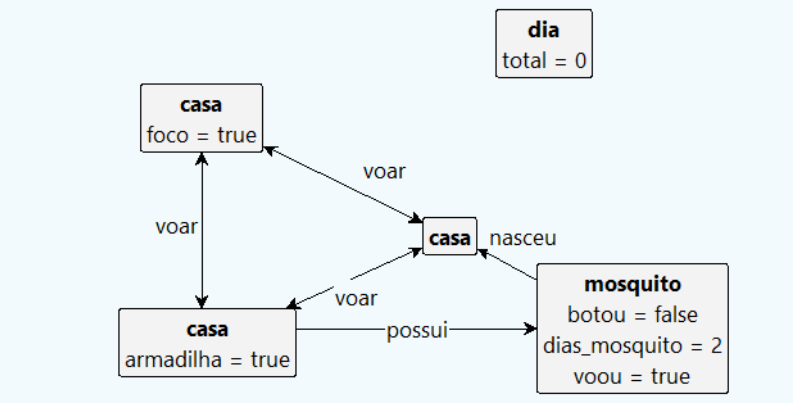

Figura 34 – Cenário Transformado de Exemplo - Groove

Um exemplo de regra de transformação na visualização é feito pelo comportamento de "Vôo" em que o nó "possui" deixa de estar ligado a um nó "casa" (Figura 33), para se ligar a outro nó "casa" (Figura 34), incrementando também o número de dias do mosquito (vida).

3.8 Simulação Orientada a Regras e Situações

A terceira implementação foi feita utilizando o *Scene* (PEREIRA; COSTA; ALMEIDA, 2013). O *Scene* é uma plataforma que faz uso de situações e de processamento de eventos complexos (*Complex Event Processing - CEP*) para especificar ocorrências de situações e eventos na simulação. Informações mais detalhadas sobre o *Scene* estão presentes no Referencial Teórico (Seção 2.4).

```
import org.kie.api.definition.type.Expires;
import org.kie.api.definition.type.Role;

@Role(Role.Type.EVENT)
@Expires("38d")
public class Mosquito{
   private House house;
        public House getHouse() {
                return house;
        }
        public void setHouse(House house) {
                this.house = house;
        }
}
```

Figura 35 – Mosquito Tratado como Evento

A estrutura de objetos da implementação do *Scene* é a mesma do *Java*, sendo utilizados os objetos "*House*" (Figura 10), "*Eggs*" (Figura 11), "*Mosquito*" (Figura 12), "*Agents*" (Figura 13) e "*Scenary*" (Anexo A). A diferença, no entanto, é que o objeto "*Mosquito*" agora é tratado como um evento (Figura 35) contendo um tempo de expiração (tempo de vida) e automaticamente é retirado da *Working Memory* após sua expiração de 38 dias utilizando a notação "*@Expires("38d")*".

A simulação no *Scene* utiliza regras, eventos e situações para a composição. As regras, descritas no *drl* do *Scene*, verificam uma determinada condição e aplicam o comportamento que a simulação deve realizar. Os eventos, criados a partir do *Java* com notações em *Drools*, são utilizados em episódios e em objetos que possuem uma expiração. As situações são úteis na identificação de determinados comportamentos que incluem uma ativação e desativação.

```
@Role(Role.Type.EVENT)
@Expires("1d")
public class mosquitoFlown {
    private Mosquito migrated;
    public Mosquito getMigrated() {
        return migrated;
    }
    public void setMigrated(Mosquito migrated) {
        this.migrated = migrated;
    }
}
```

Figura 36 – Evento “mosquitoflown”

“*Mosquitoflown*” (Figura 36) é um evento “marcador”, que indica se o mosquito (atrelado ao evento) voou ou não em um período de tempo de um dia.

Para o vôo do mosquito, *Scene* faz a detecção da regra “mosquitoFlying” (Anexo B), verificando se há o evento “*mosquitoFlown*” (Figura 36) para cada objeto “*mosquito*”. Se não houver, um novo evento é criado, o “*mosquito*” muda “*house*” para outro objeto “*house*” (assim como no *Java*) representando o vôo do mosquito. O evento “*mosquitoflown*” é inserido na *Working Memory* e, então, “*mosquito*” é atualizado na *Working Memory*.

```
@Role(Role.Type.EVENT)
@Expires("20d")
public class mosquitoLayedEggs {
    private Mosquito layed;
    public Mosquito getLayed() {
       return layed;
    }
    public void setLayed(Mosquito layed) {
        this.layed = layed;
    }
}
```

Figura 37 – Evento “mosquitoLayedEggs”

O evento “*mosquitoLayedEggs*” (Figura 37) é um indicador que os ovos foram ovipositados, contendo uma expiração de 20 dias. O evento indica que determinado mosquito (atrelado ao evento) já fez a oviposição.

```
@Role(Role.Type.EVENT)
@Expires("20d")
public class eggsHatched {
    private Eggs hatched;
    public Eggs getHatched() {
        return hatched;
    }
    public void setHatched(Eggs hatched) {
        this.hatched = hatched;
    }
}
```

Figura 38 – Evento "eggsHatched"

O evento "*eggsHatched*" (Figura 38), assim como "*layedeggs*", indica que os ovos foram depositados pelo mosquito e que estão nascendo no período de 20 dias, com a sua expiração.

Para o comportamento de oviposição por parte do mosquito (regra "mosquitoLayingEggs" - Anexo B), *Scene* verifica se "*House*" possui o atributo "*trap*" (indicando que a casa possui uma armadilha) ou "*activefocus*" (indicando que a casa é um foco) verdadeiro. Uma vez que o evento "*mosquitoLayedEggs*" não existe para o mosquito (indicando que o mosquito não fez a oviposição), o evento é então criado e ovos são inseridos. O evento "*eggsHatched*" também é criado para indicar o tempo de vida dos ovos.

Quando o evento "*eggsHatched*" (Figura 38) em uma "*house*" com "*activefocus*" verdadeiro (casa com focos) ultrapassa 20 dias de existência na *working memory*, 10 mosquitos ("*Mosquito*") são criados e adicionados na simulação, removendo também os ovos ("*eggs*") que deram origem aos mosquitos. Este comportamento pode ser observado na regra "eggshatching" (Anexo B).

Um outro comportamento é o de captura do mosquito pela armadilha. Há a regra "capturedMosquitoAndCallingAgent" (Anexo B) que ativa a situação de captura do mosquito pela armadilha, verificando-se que existem ovos ("*eggs*") em uma casa com armadilha (*House(trap==true)*) e que os ovos estão sendo chocados por 4 dias ("*eggsHatched(this.hatched==eggs) over window:time(4d)*").

A partir da ativação da situação "*capturedMosquitoAndCallingAgent*" (Anexo B), a regra "*mosquitoCaptured*" (Anexo B) é detectada, fazendo a remoção do mosquito capturado pela armadilha da estrutura (em *Java*) e da *working memory*.

Quando há uma desativação da situação "*capturedMosquitoAndCallingAgent*" (Anexo B), que por sua vez possui duração de 4 dias, o agente de saúde é então criado na regra "AgentWorking" (Anexo B). Assim como em *Java* (Anexo A), o agente verifica as listas de ovos e mosquitos da casa em que está presente. Uma vez na casa, o método "*clear()*" retira todo os ovos. Caso sejam encontrados mosquitos, os mosquitos são então retirados da *working memory* (método "*retract(mosquito)*"). Também, os focos da casa em que está são combatidos, modificando o booleano "focus" através do método "*setActivefocus(false)*". Todo o comportamento é realizado em 5 outras casas, escolhendo-se aleatoriamente casas

vizinhas ("*neighborHouse*").

O clima (simulação de chuva) é simulado a partir de um evento: "*Rain*" (Anexo B). Quando não houver o evento (a cada 15 dias de expiração na *working memory*), um novo evento é criado, retornando verdadeiro ao booleano "*focus*" dos objetos "*House*" da simulação.

Assim como no *Java*, para se realizar a visualização da simulação há o retorno de mensagens sobre ocorrências de situações e eventos na simulação além de mostrar o total de mosquitos e ovos.

3.9 Simulação Orientada a Agentes

Uma outra implementação foi realizada com o *RePast*, especificamente na versão *ReLogo* na qual os desenvolvedores devem definir os tipos de agentes (indivíduos) e seus comportamentos. As informações sobre o *RePast* estão presentes na Seção 2.2.2.

A execução da simulação no RePast é realizada através de passos de tempo, ou "*ticks*", no qual cada agente deve ser definido em relação a um "*tick*" ou um espaço de tempo. O *RePast* fornece diversas funções prontas espaciais em relação ao comportamento dos agentes, isso é, o *RePast* possui em sua biblioteca métodos para movimentação dos agentes e métodos matemáticos para a simulação tais como "mover para determinado ponto" ou "calcular um ângulo para movimento". As simulações também requerem a conFiguração do componente "*UserObservation*", que oferece uma interface gráfica para a conFiguração do cenário (definição de cores e formas dos agentes).

A seguir serão apresentados os agentes que compõem a simulação do Aedes aegypti.

```
class House extends ReLogoTurtle {
    def focus = false
    def trap = false
    def activefocus = false
```

Figura 39 – Agente da Casa

Inicialmente o agente "*House*", representa a casa e contém três atributos, sendo eles o "*focus*" que indica se a casa é um foco para deposição de ovos, "*trap*" que apresenta se há armadilhas ou não na casa e "*activefocus*" que indica se o foco está ativo.

```
class Eggs extends ReLogoTurtle {
    def days=0
```

Figura 40 – Agente de Conjunto de Ovos

O agente "*eggs*", representando os ovos da simulação, inclui apenas o atributo de dias, que especifica o tempo de vida dos ovos na simulação.

```
class Mosquito extends ReLogoTurtle {
    def days = 0
    def x = this.getXcor()
    def y= this.getYcor()
    def housecanfly
```

Figura 41 – Agente do Mosquito

O agente "*Mosquito*" contém o atributo dias de vida ("*day*"), especificando o número de dias de vida em que está na simulação, assim como os ovos. Também, dois atributos de posição em em pixels da simulação ("X" e "Y") estão presentes pois o mosquito é um agente dinâmico, isso é, pode sofrer mudança em sua posição na simulação ao decorrer dos dias. Além disso, há um atributo com as casas possíveis de vôo ("*housecanfly*").

```
class Agents extends ReLogoTurtle {
    def calling = 0
```

Figura 42 – Agente de Conjunto de Agentes

O agente de saúde "*Agents*" possui apenas o atributo contador de dias em que foi chamado ("*calling*") sendo este atributo o responsável pela passagem do tempo ao decorrer dos dias e efetivamente o agente começar seu trabalho no terceiro dia.

A seguir serão apresentados os comportamentos em razão dos *steps*, ou passos, dos agentes na simulação.

```
def step(){
    //choose another house and lay eggs if mosquito has less than 38 days.
    housecanfly = maxOneOf(inRadius(houses(), 100)){
        count(inRadius(houses(), 100))
    }
    while(housecanfly.distancexy(x, y)>100)
    {
        housecanfly = maxOneOf(inRadius(houses(), 100)){
            count(inRadius(houses(), 100))
        }
    }
    if(days<38)
    {
        face(housecanfly)
        moveTo(housecanfly)
        days++
        if(housecanfly.activefocus==true)
        {
            hatchEggss(1)
        }
        if(housecanfly.trap==true)//mosquito die and agent is called
        {
            die()
            hatchAgentss(1)
        }
    }
    else
    {
        die()//mosquito die after 38 days
    }
}
```

Figura 43 – Comportamento do Mosquito

Inicialmente, é escolhida uma casa aleatória no raio de 100 metros na simulação em relação ao ponto da casa inicial (“*maxOneOf(inRadius(houses(),100))*”). Então, se a idade do atributo do agente mosquito for menor que 38 (“*days<38*”), o agente mosquito se direciona à casa (“*face(housecanfly)*”), move até a casa (“*moveTo(housecanfly)*”), incrementa sua própria idade (“*days++*”) e verifica a possibilidade de oviposição. Se o mosquito está em uma casa com foco ativo (“*housecanfly.activefocus==true*”), um conjunto de agente ovos é colocado na simulação (“*hatchEggss(1)*”). Se a oviposição for feita em uma casa com armadilha (“*housecanfly.trap*”), então o agente mosquito em questão morre (“*die()*”) e agentes de saúde são colocados naquele ponto (“*hatchAgentss*”). Caso a idade do agente mosquito seja de 38 ou mais, o agente morre.

```
def step(){
    days++
    if(days==20) {
        hatchMosquitos(10)//10 mosquitos born after 20 days
        die()
    }
}
```

Figura 44 – Comportamento dos Conjuntos de Ovos

Para o comportamento dos agentes de conjunto de ovos, inicialmente é feito o incremento da idade de cada agente ovos (*“days++”*) e há a comparação se o agente ovos possui 20 dias de vida (*“days==20”*). Caso possua, o agente morre e dá origem a 10 agentes “*Mosquito*”.

```
def step(){
    calling++
    if(calling==3)
    {
        def counting = 5
        agentssHere().removeAll()
        while(counting>0)//visiting houses and removing eggs/mosquitos/foci
        {
        def houseat = oneOf(housesHere())
        if(houseat!=null)
        {
        desinfect(houseat)
        }
        mosquitosHere().removeAll()
        eggssHere().removeAll()
        def next = maxOneOf(neighbors()){
            count(housesOn(it))
        }
        moveTo(next)
        counting--
        }
        die()
    }
}

def desinfect(House){
    House.activefocus=false
}
```

Figura 45 – Comportamento dos Agentes de Saúde

O agente "*Agents*", representando os agentes de saúde na simulação, incrementa no início seu atributo "*calling*" significando que precisa passar 2 dias de sua primeira chamada para começar a atuar (simulando o tempo de chamada de um agente de saúde a partir da armadilha). Ao terceiro dia, o agente faz a visita em 5 casas, removendo mosquitos, ovos e desinfetando (mudando o "*activefocus*" para falso).

```
ask (housess())
{
    if(contador%15==0)
    {
        step()
    }
}
```

Figura 46 – Comportamento de Chuva

```
def step(){
    if(focus==true)
    {
        activefocus=true
    }
}
```

Figura 47 – Comportamento de Chuva na Casa

O comportamento do tempo é realizado direto na parte de chamada dos *steps* dos agentes, sendo executado apenas quando o dia (através do atributo "contador") possuir um resto de divisão por 15 igual a 0. Ou seja, de 15 em 15 dias, o *step* é executado, fazendo com que o "*activefocus*" volte a ser verdadeiro, simulando a chuva.

Cada entidade de agente no *RePast* é configurada de forma a exibir uma figura em um espaço no plano (X e Y). Dessa forma, os mosquitos, ovos e agentes possuem localização própria e mudam sua localização além de serem deletados ou surgirem novos ao decorrer da simulação.

3.10 Simulação Orientada a Interfaces Gráficas

O *Processing* foi utilizado para a simulação orientada a interfaces gráficas e maiores informações sobre o *Processing* estão presentes no referencial teórico (Seção 2.3).

O desenvolvimento da simulação envolve tipicamente a definição do visual e das operações (mudanças nos atributos) que farão a execução do comportamento da simulação. Apesar da simulação no *Processing* apresentada aqui seguir um comportamento baseado em agentes (assim como o *RePast*), o *Processing* possibilita a realização de outros tipos de simulações, tais como simulações 3D, celulares, dentre outras.

A simulação implementada no *Processing* segue o mesmo estilo de simulação do *RePast*, sendo desenvolvida através de agentes que possuem um comportamento próprio

e autônomo na simulação. A seguir serão apresentados os agentes que fazem parte da simulação.

```
class House {
float X;
float Y;

boolean focus = false;
boolean trap = false;
boolean activefocus = false;
```

Figura 48 – Agente da Casa

O agente "House" (Figura 48), que representa a casa, contém duas variáveis "X" e "Y" indicando sua posição no espaço de simulação em *pixels* e três booleanos de controle, para a definição dos focos, armadilha e focos ativos ("*focus*", "*trap*" e "*activefocus*" respectivamente).

Os conjuntos ovos na simulação são representados pelo agente "*Eggs*" e variáveis de posição no espaço em pixels "X" e "Y" da simulação. Além disso, possuem um inteiro de controle de dias de vida "*days*".

O agente "*Mosquito*", representando o mosquito, possui os mesmos atributos do agente "*Eggs*", sendo eles dois *floats* de posição no espaço e um inteiro de controle representando os dias de vida.

O agente "*Agents*", representando os agentes de saúde, possui os mesmos atributos de posição e vida assim como "*Eggs*" e "*Mosquito*".

A seguir serão apresentados os comportamentos em passos (steps) da simulação.

No comportamento do mosquito (Anexo C) inicialmente há uma lista de possibilidades na qual o mosquito pode voar. É feito um cálculo de distância entre o mosquito e a casa na qual ele pode voar (utilizando o método "*distance(X,Y,X2,Y2)*" e o resultado desse cálculo (as casas em que pode voar) entra na lista de possibilidades ("*possibility*"). Então, uma casa é escolhida aleatoriamente com o método "*random*". Uma vez escolhida a casa e tendo o mosquito uma idade menor que 38 ("*days<38*"), o mosquito modifica sua coordenada X e Y e incrementa o número de dias. Se na casa escolhida houver um foco (método "*getActiveFocus()*"), há a adição de ovos naquela posição. Se houver uma armadilha (método "*getTrap()*"), há a adição de agentes naquela posição e o mosquito sai da simulação. Caso os dias de vida ultrapassem 38, o mosquito também sai da simulação (método "*die*").

```
void step()
{
  days++;
  if (days==20)
  {
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    mosquitos.add(new Mosquito(X, Y));
    this.die();
  }
}
```

Figura 49 – Comportamento dos Ovos

Para o comportamento do conjunto de ovos (Figura 49), uma vez que o conjunto chega ao dia 20 ("*days*" igual a 20), 10 mosquitos são adicionados no mesmo local e o agente representando os ovos, "*eggs*", é retirado da simulação.

O comportamento dos agentes de saúde (Anexo D) contém um *step* em que os agentes mantém um contador de dias, que é incrementado a cada step ("*days++*"). No terceiro dia, a atuação do agente começa. Na atuação, o agente faz com que a casa não tenha mais focos ativos ("*removeAllHere()*"), ovos ("*removeAllEggsHere()*") ou mosquitos ("*removeAllMosquitosHere()*"). Em seguida, o agente escolhe uma nova posição a partir de uma lista de possibilidade de atuação em casas vizinhas. Escolhida a casa vizinha para atuação, o agente repete o processo por mais 5 vezes.

```
if (tick%15==0)
{
  for (int i = 0; i < houses.size(); i++) {
    House element = houses.get(i);
    if (element.getFocus()==true)
    {
      element.setActivefocus(true);
    }
  }
}
```

Figura 50 – Comportamento de Chuva

O tempo é verificado diretamente na execução da simulação (Figura 50), não sendo definido em nenhum step. Há a comparação do *tick* em que a simulação se encontra, verificando se está no dia 15 e posteriormente há a mudança de todos os elementos "*House*" para focos ativos.

Assim como no *RePast*, há a configuração visual do cenário em que as entidades (mosquitos, ovos e agentes), com uma figura representativa em sua localização, sofrem modificações ao longo da simulação.

Figura 51 – Cenário Inicial de Exemplo - Comportamento de Vôo

Figura 52 – Cenário Modificado de Exemplo - Comportamento de Vôo

A Figura 51 demonstra um cenário inicial com três casas e um mosquito na casa superior. Ao sofrer modificações de comportamento de vôo, o mosquito encontra-se posteriormente em outra casa, como na Figura 52.

3.11 Análise das Simulações

Com a implementação do cenário do *Aedes aegypti* em cinco diferentes ferramentas, é possível fazer uma análise comparativa das ferramentas. São utilizados como parâmetros de comparação para avaliar as ferramentas:

- A análise que a ferramenta possibilita fazer ao decorrer da simulação - As informações que a ferramenta fornece ao usuário durante a execução que facilitam a identificação de comportamentos e dados úteis a respeito da simulação.
- O desempenho da ferramenta em relação ao tempo e à memória - A execução do estudo de caso em um ambiente controlado fornece uma identificação das ferramentas com melhor desempenho através da comparação do tempo de execução e do número máximo de estruturas suportadas na memória.
- A linguagem de programação e sua organização - As funções, documentações e detalhes específicos da linguagem de programação.

3.11.1 Comparação das Análises de Simulação

Java dá a liberdade ao programador de escolher as informações a respeito da simulação que devem ser exibidas ao usuário. Ao mesmo tempo que essa liberdade é boa, torna-se mais difícil escrever o que exibir e como exibir. Sem a utilização de nenhuma biblioteca, cabe ao programador escolher como será a saída da informação (*log*, *print*, ...) e quais informações serão exibidas (através de *if-then-else*). Para a simulação do *Aedes aegypti*, apenas informações a respeito da quantidade de mosquitos e ovos foram exibidas.

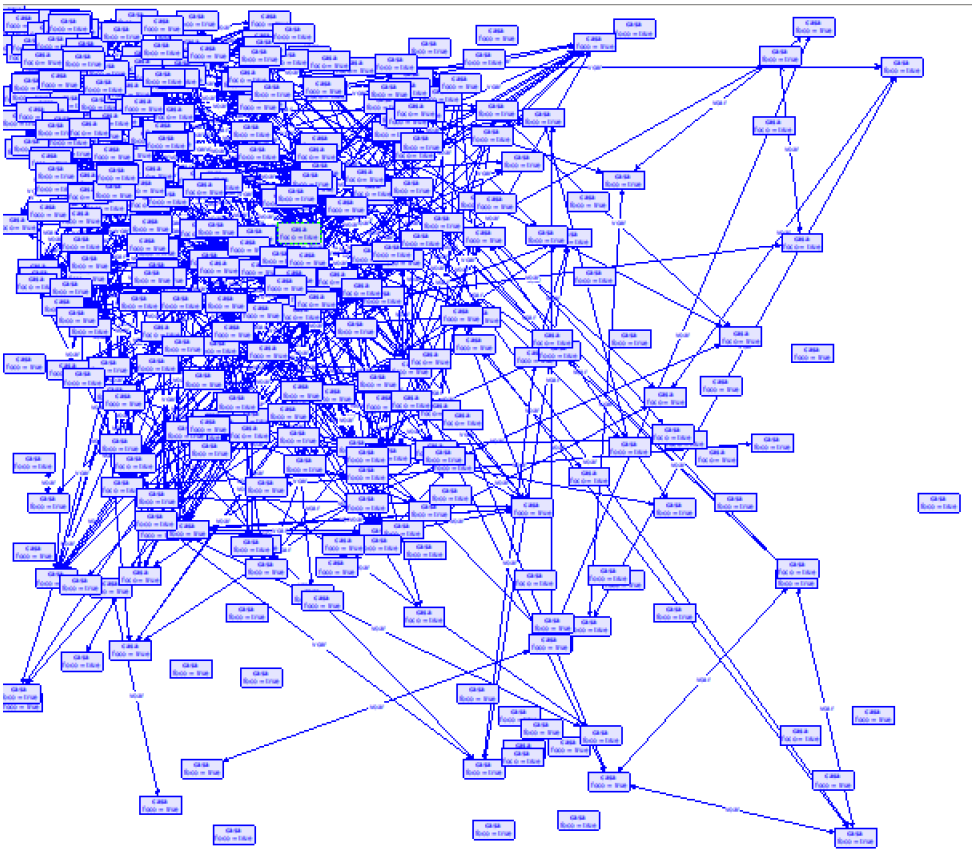

Figura 53 – Cenário Inicial do *Aedes Aegypti* no *Groove*

Groove provê uma análise de simulação que utiliza a visualização de grafos durante o processo de transformação. Os grafos são transformados e exibidos em uma área de visualização. Apesar do recurso de observação, é difícil verificar estruturas quando o cenário de simulação possui um grande tamanho, como por exemplo no cenário do *Aedes aegypti* (Figura 53) em que é impossível de se identificar um mosquito em uma casa.

```
rule "capturedMosquitoSevereSituation"
@role(situation)
@type(capturedMosquitoSevereSituation)
        when
                $house: House()
                $situation: Number(this>3) from accumulate($sit:
                    capturedMosquitoAndCallingAgent
                    (house==$house, active==true), count($sit))
        then
                SituationHelper.situationDetected(drools);
end
```

Figura 54 – Situação Severa do Mosquito

Scene por sua vez já utiliza o conceito de situação para analisar a simulação. Com este conceito, as informações de interesse são observadas de forma similar ao que acontece na vida real, através de padrões de fatos que ocorreram ao longo do tempo. Um exemplo é a situação severa do mosquito, na qual mosquitos são capturados de forma contínua por

uma armadilha (Figura 54). Também, no *Scene* é possível a utilização do *Java* para uma análise quantitativa da simulação (número de estruturas).

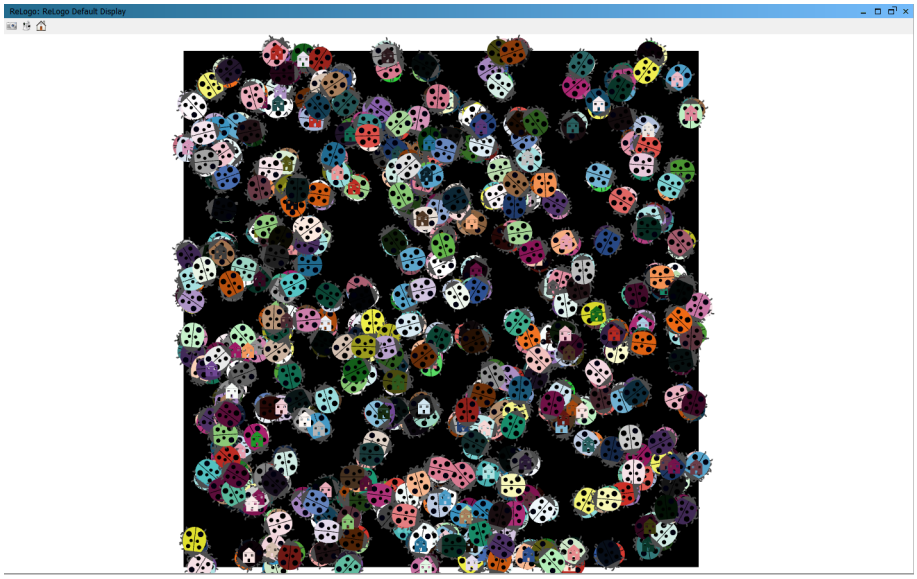

Figura 55 – Cenário Inicial do *Aedes aegypti* no *ReLogo*

O *ReLogo* provê uma área para visualização da simulação através de formas com cores e uma área com ferramentas para análise quantitativa das estruturas da simulação. Assim como o *Groove*, em cenários com muitos agentes (como o cenário inicial do *Aedes aegypti* na Figura 55) há a dificuldade na identificação das ocorrências na simulação. Apesar disso, a dificuldade de visualização é menor em relação ao *Groove* em razão das cores e formas definidas previamente sobre o agente.

Figura 56 – Cenário Inicial do *Aedes aegypti* no *Processing*

O *Processing* possibilita a análise quantitativa da mesma forma que o *Java* e, ao mesmo tempo, uma análise visual como o *ReLogo*. No entanto, pela liberdade de escolha das formas, cores e tamanhos que compõem o visual de simulação, o *Processing* sobressai

no quesito de visualização de simulações, sendo mais claro na exibição da simulação pois a visualização é totalmente conFigurada ao gosto do implementador, como no cenário inicial do *Aedes aegypti* (Figura 56) em que a área de visualização foi conFigurada para ser extensa e representar fielmente a localização das casas na vida real (longitude e latitude).

3.11.2 Comparação de Desempenho

Utilizando o cenário inicial do Aedes aegypti, com uma população de 10 mosquitos em cada uma das casas e 17 armadilhas espalhadas, foi realizado um estudo de caso para comparação do desempenho das ferramentas através de uma média entre 10 execuções de um mesmo cenário de entrada para a simulação. Os resultados foram obtidos em um *Macbook Pro* (Retina, 13 polegadas, modelo "*Early 2015*") com um processador Intel i5-5257U 2.7 Ghz, memória *RAM* de 8GB 1867 MHz DDR3, 128GB de armazenamento *SSD* e sistema operacional *Mac OS High Sierra* 10.13.3.

Tabela 23 – Tempo de Execução em Milissegundos

Dias	**Java**	**GROOVE**	**Scene**	**RePast**	**Processing**
5	22	48224	5783	49038	287
10	28	861846	8311	92073	582
15	33	–	9854	122607	953
20	40	–	11288	146686	1408
25	76	–	107747	1039921	4120
30	115	–	488994	3445092	13642
35	206	–	957218	6642166	33269
40	254	–	1782615	10276445	63996
50	822	–	–	–	1065296
60	7235	–	–	–	15496139
70	330112	–	–	–	–
80	33757062	–	–	–	–

A Tabela 23 mostra o resultado dos testes em razão dos dias de simulação do cenário de 5 a 90 dias em um tempo de execução em milissegundos. A simulação revelou que a estratégia de combate dos agentes de saúde não foi suficiente para inibir a proliferação dos mosquitos, com isso, as estruturas de mosquitos e ovos sofreram um aumento exponencial, levando ao uso de uma grande quantidade de memória e, posteriormente, a parada do programa em função da falta de memória. Dessa forma, as linhas que não possuem resultado de tempo (linhas com –) representam um resultado no qual o *software* não conseguiu atingir.

Tabela 24 – Número Máximo de Elementos na Simulação

	Dias	***Mosquitos***	***Ovos***
Java	82	37118209	232602309
Groove	10	665	8097
Scene	42	78129	309638
RePast	43	79359	317728
Processing	64	17757259	95655560

Já a Tabela 24 demonstra o número máximo de elementos que o *software* conseguiu suportar antes de ocupar toda a memória do computador e sofrer uma interrupção. Caso o combate ao mosquito fosse adequado pelos agentes de saúde, o número de mosquitos e ovos na simulação estaria estável ou diminuiria com o passar dos dias.

Dessa forma, podemos observar neste estudo de caso que o *Groove* apresentou o pior resultado entre as ferramentas pelo fato de que é um *software* de uso genérico, para a exploração do espaço de estados em sistemas. O *Java* apresentou o melhor resultado, seguido pelo *Processing*. *Scene* e *RePast* obtiveram um resultado semelhante no número máximo de elementos na memória, porém, *Scene* obteve um tempo menor na execução, desde o início.

3.11.3 Comparação do Tipo de Programação

Java é uma linguagem orientada a objeto bastante utilizada com uma vasta documentação online. No entanto, a utilização do *Java* puro (sem nenhuma biblioteca para ajuda) pode tornar mais difícil o processo de elaboração de simulações. Não há métodos prontos para auxiliar na implementação, não há suporte ao gerenciamento de simulações e a parte de análise da simulação também deve ser implementada manualmente (não há métodos para exibição de informações a respeito da simulação). Dessa forma, a manutenção e o entendimento de uma simulação implementada em *Java* se torna difíceis e complexas à medida que a simulação recebe novas funcionalidades e informações.

Groove provê uma programação visual: nós e arestas são usados na composição de regras e ações em uma simulação. No próprio *software* há uma documentação interativa para auxiliar o desenvolvedor nas diversas funções que oferece. Apesar disso, a elaboração de um cenário inicial (como o do *Aedes aegypti*) pode demandar tempo e paciência caso feita manualmente. A implementação de uma simulação no *Groove* possui um fácil entendimento, execução e manutenção.

Scene possui suporte nativo ao gerenciamento de situações, regras e eventos, sendo uma maneira mais próxima ao natural de implementar simulações devido ao alto nível de abstração que regras, eventos e situações proporcionam em relação ao mundo real. *Scene* é o único *software* que possui suporte explícito ao gerenciamento de situação de

conhecimento do autor até o momento da escrita desta dissertação.

RePast é uma ferramenta com propósito específico na elaboração de simulações e, por conta disso, provê diversas funções e métodos prontos, facilitando o trabalho de implementação por parte do implementador, a manutenção da simulação e o entendimento do que a simulação propõe a fazer. As funções e métodos prontos são inteiramente documentados em seu site oficial, possuindo um excelente suporte na elaboração de simulações.

Processing é uma linguagem de programação de propósito genérico e, apesar disso, a implementação de uma simulação baseada em agentes foi simples de ser concluída em razão da qualidade da documentação disponível e das funções e métodos prontos do *Processing*. Todo o aspecto da programação pode ser mudado no *Processing*, permitindo uma maior liberdade e adequação às necessidades do implementador, tendo este uma liberdade de utilizar outros tipos de abordagens na simulação, tais como visualizações em 3D ou celulares. Além disso, por ser uma linguagem de alto nível e mais fácil de ser aprendida (em razão do objetivo principal da linguagem), a simulação facilita a inclusão de novas funções, entendimento dos códigos e execução de novos cenários de simulação.

Assim, adquiriu-se um conhecimento a respeito das plataformas de simulação com a implementação de um cenário do Aedes aegypti. Tal conhecimento foi utilizado para a composição da plataforma *ProScene*, a ser apresentada no próximo capítulo.

4 *ProScene* - Plataforma de Integração

Com o conhecimento adquirido na implementação de várias simulações, em diferentes tecnologias (Capítulo 3), chegou-se à conclusão que a ferramenta *Processing* oferece vantagens no desenvolvimento de simulações que definam explicitamente o conceito de situação (*Simulações Situation-Aware*), como por exemplo, realizar estudos de fenômenos complexos em diversos contextos, oferecer suporte ao desenvolvedor na observação de situações ocorridas ao longo da simulação e definir e analisar a simulação em função do comportamento (Seção 2). Portanto, a fim de incorporar aspectos de *Situation Awareness* a *Processing*, foi idealizada a plataforma de integração denominada *ProScene*. O objetivo final desta ferramenta é (i) facilitar o desenvolvimento de *Simulações Situation-Aware* e (ii) permitir o monitoramento do ciclo de vida dessas situações por meio de interfaces gráficas. Este capítulo discute o projeto dessa plataforma (arquitetura conceitual), sua implementação, bem como diretrizes para implementação de novas simulações utilizando *ProScene*.

A composição do *ProScene* visa a obtenção de conjunto das melhores ferramentas com análise, desempenho e tipo de programação balanceados. Dessa forma, as ferramentas que se destacaram foram *Scene*, *Processing* e *RePast*.

Foi observada que a análise do *Scene* com Situações apresenta uma forma mais realista de observar os acontecimentos ao decorrer da simulação, melhorando o entendimento do contexto das simulações. O *Processing* possui diversas funções prontas que facilitam a implementação por parte do programador além de apresentar a maior liberdade na elaboração do visual da simulação. O *RePast* e sua programação focada em agentes apresentou-se como uma proposta organizada e simples de definir ações às entidades que compõem a simulação.

Assim, escolhidas as três ferramentas para a composição do *ProScene*, foi captado o melhor aspecto de cada uma das ferramentas, sendo o gerenciamento de situações do *Scene*, a possibilidade de construção de visualização de simulações com a facilidade de funções prontas do *Processing* e a forma como as simulações são programadas em razão de agentes do *RePast*. Um maior detalhamento da composição do *ProScene* é apresentado a seguir.

Neste capítulo serão apresentados os requisitos, arquitetura conceitual, arquitetura de implementação e as diretrizes para implementação.

4.1 Requisitos

O projeto da arquitetura conceitual de *ProScene* segue os seguintes requisitos:

- Permitir a criação dos agentes que compõem a simulação - A elaboração de simulações utilizando agentes é facilitada do ponto de vista estrutural de uma simulação, como visto na implementação da simulação do *Aedes aegypti* com o *software RePast* (Seção 3.9). Os agentes são os indivíduos presentes que sofrem modificações durante a execução e dão "vida" às simulações.

- Possibilitar a definição do comportamento dos agentes de simulação - O comportamento dos agentes, definido por *steps*, facilita a definição das ações comportamentais que os agentes efetuarão ao longo da simulação. O comportamento dos agentes é responsável pela "vida" de cada um, cumprindo ordens a partir das definições dos *steps*;

- Criar e identificar os eventos que ocorrem durante o comportamento dos agentes de simulação - Os eventos, como os auxiliados pelo *Scene* na simulação do *Aedes aegypti* (os mosquitos que migraram foram marcados em eventos de migração, por exemplo - Seção 3.8), facilitam a identificação de determinados padrões na simulação e melhoram o entendimento do ocorrido. Ao utilizar eventos, o programador trabalha de uma forma mais direta com as ocorrências de determinados comportamentos na simulação, sem precisar trabalhar com a estrutura dos agentes (atributos dos agentes), identificando os comportamentos desejados diretamente de acordo com a ocorrência de eventos;

- Permitir a definição das situações que serão monitoradas na simulação - As situações são interessantes de serem monitoradas e identificadas pelo programador, sendo um método de abstração com um nível mais próximo ao que acontece na vida real e possibilita a percepção dos elementos e eventos em relação ao tempo e espaço;

- Viabilizar a escolha de formas e cores que identificam o cenário simulado e as ativações / desativações de situações - promover um método que faça o uso de formas e cores que tenham relação com a gravidade da situação e com o cenário em si;

- Visualizar a execução da simulação e identificar visualmente os agentes e as situações que foram ativadas ou desativadas ao decorrer da simulação, incluindo sua localização - Possibilitar ao programador a visualização do momento de ativação / desativação de uma situação de interesse;

4.2 Arquitetura Conceitual

A partir dos requisitos, a arquitetura conceitual proposta é apresentada na Figura 57.

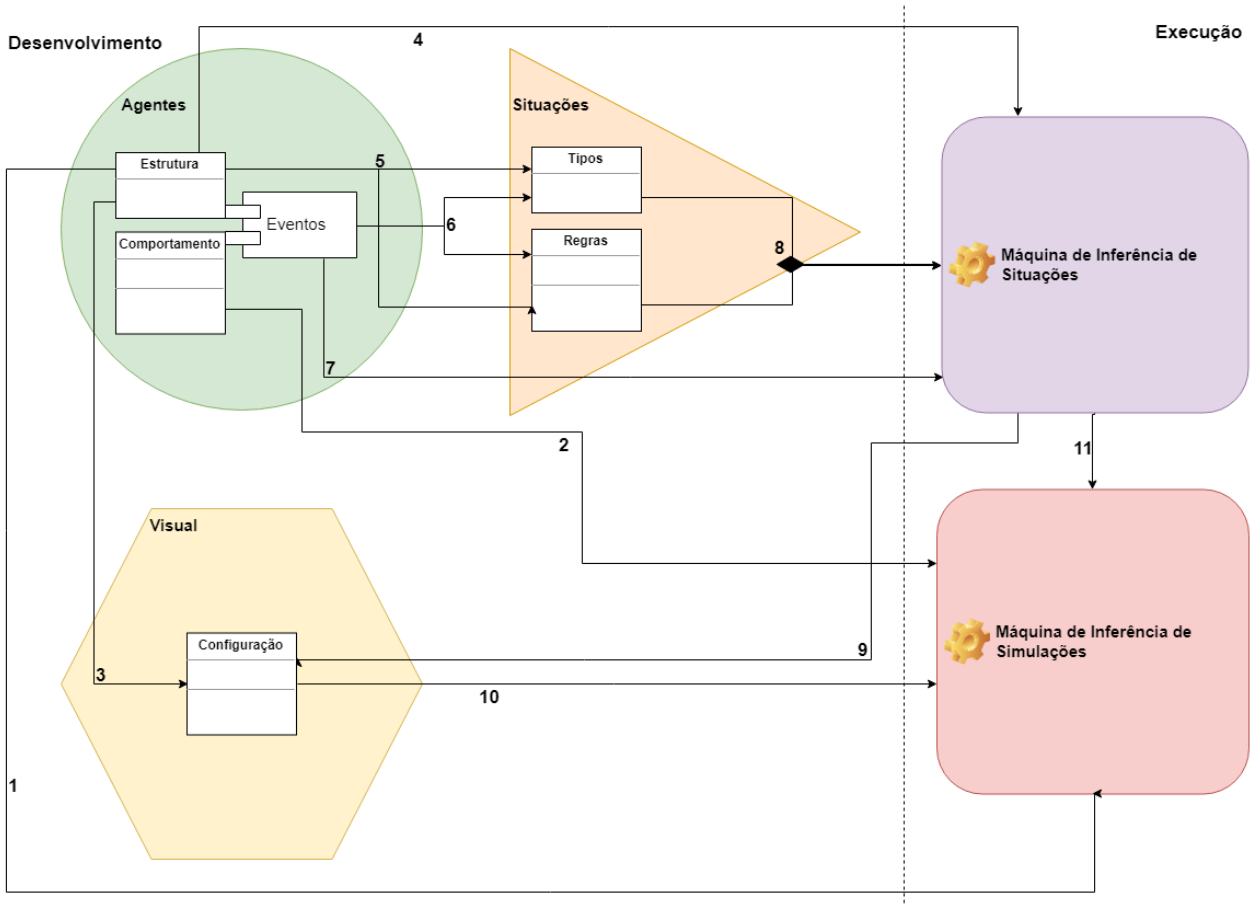

Figura 57 – Arquitetura Conceitual.

A parte redonda esverdeada da Figura 57 representa o componente de agentes com seus respectivos códigos, representados pelos quadrados brancos. A parte alaranjada triangular da Figura 57 representa o componente de Situações e seus respectivos códigos, representados pelos quadrados brancos. A parte hexagonal amarelada representa o componente Visual e seu respectivo código, representado pelo quadrado branco. As máquinas de inferência são representadas pelos quadrados com cantos arredondados.

Ao lado esquerdo da Figura 57 pode-se observar os componentes que devem ser implementados, sendo eles: os **agentes**, as **situações** e o **visual** da simulação em códigos.

Os **agentes** são as entidades ou estruturas que fazem parte do cenário de simulação, dessa forma, as **estruturas** (atributos do agente) sofrem transformações durante a simulação. Essas transformações são definidas pelo **comportamento** (o que o agente deverá fazer na simulação) previamente programado pelo implementador, definido por *steps* ou passos no qual cada agente deve seguir autonomamente. Cada comportamento pode

gerar um ou mais **eventos**, ou seja, pode gerar acontecimentos interessantes auxiliando na identificação de determinados comportamentos (incluindo a detecção de situações com esses eventos) e melhorando o entendimento do cenário de simulação.

Já as **situações** englobam os **tipos** de situações que podem ocorrer e suas respectivas **regras** para a detecção de determinados acontecimentos para a situação se manter ativa, sendo assim o programador deve criar um tipo de situação e programar as regras que a detectam e a mantém ativa.

A parte **visual** da simulação engloba a implementação da **conFiguração** do cenário, isso é, qual a aparência do cenário de simulação na execução das simulações.

Ao lado direito da Figura há a parte de execução da simulação, ou seja, os mecanismos e tecnologias computacionais que fazem a execução da simulação, sendo eles: a **máquina de inferência de situações** e a **máquina de inferência de simulações**.

A **máquina de inferência de situações** é responsável pela detecção de situações na simulação e pela manutenção dos objetos e eventos na memória.

A **máquina de inferência de simulações** é responsável pela modificação das estruturas dos agentes através da execução do comportamento. Também, esta máquina é responsável pela exibição do ambiente de simulação.

A seguir serão apresentados os relacionamentos (arestas) da Figura 57:

1. As estruturas dos agentes devem ser adicionadas à máquina de inferência de simulações, para que as mesmas sejam modificadas ao decorrer da execução da simulação.

2. O comportamento dos agentes também deve ser adicionado à máquina de inferência de simulações, para que sejam dadas as instruções de modificações das estruturas, ou seja, o código para que os agentes executem funções na simulações (tenham “vida”).

3. As estruturas dos agentes também devem ser adicionadas ao código visual, fazendo com que os agentes sejam exibidos corretamente no processo de simulação. Os atributos dos agentes, por exemplo, podem ser usados para a diferenciação de um agente entre a população de agentes na execução da simulação.

4. As estruturas dos agentes são adicionadas à máquina de inferência de situações para que a máquina de inferência reconheça determinado agente durante o casamento de algum padrão.

5. As estruturas dos agentes são utilizadas na composição dos tipos de situação e das regras de situação,para que determinado agente seja parte da composição da regra de situação e, posteriormente, seja reconhecido.

6. Os eventos são utilizados na composição dos tipos de situação e das regras de situação, assim como os agentes, para que sejam parte da composição da regra de situação e, posteriormente, o agente em questão (com o evento ativo) seja reconhecido. Os eventos são interessantes em determinados casos, em que a elaboração da regra de situação utilizando apenas as estruturas dos agentes esteja complexa ou simplesmente pelo fato de que as estruturas dos agentes são dinâmicas, mudando a todo o instante e a situação poderá não captar o desejável pelo programador.

7. Os eventos são adicionados à máquina de inferência de situações para que sejam reconhecidos em situações que fazem uso deles em suas regras.

8. Os tipos e as regras de situação são adicionados à máquina de inferência de situações para que a máquina trabalhe com os padrões e reconheça as situações.

9. A ativação ou desativação de uma situação possui um retorno a conFiguração visual, sendo tratada como um agente, possuindo uma forma e cor representativa na simulação.

10. As formas e cores de todos os agentes e situações (todo o cenário) são passados à máquina de inferência de simulações, para que sejam exibidos na simulação.

11. As situações são repassadas à máquina de inferência de simulações pois são estruturas que são tratadas como agentes na simulação, sendo possível de representá-las visualmente dessa forma.

4.3 Arquitetura de Implementação

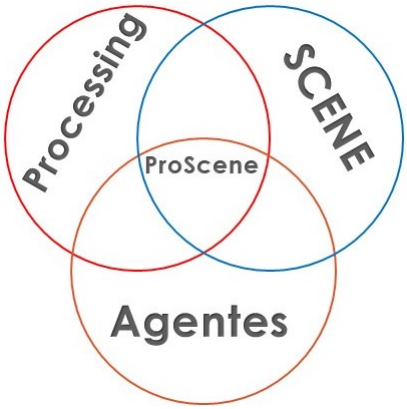

Figura 58 – Integração que representa o *ProScene*.

Conforme apresentado nos requisitos, para a implementação e integração dos diferentes conceitos é preciso utilizar tecnologias que auxiliem o implementador na elaboração

das simulações que fazem uso de Situações. Assim, a união entre as tecnologias pode ser vista na Figura 58.

ProScene utiliza o *Scene*, uma plataforma de gerenciamento de situações que possui suporte ao *Situation Awareness* (PEREIRA; COSTA; ALMEIDA, 2013). *Scene* possui o papel de monitoramento das situações ocorridas na *working memory* durante o processo de simulação. Com isso, *Scene* consegue reagir a determinados padrões de situações tanto identificando visualmente a situação ocorrida (ativação, desativação e localização da situação) quanto modificando parâmetros da própria simulação. Dessa forma, o *Scene* fundamentalmente monitora as ocorrências de situações através de sequências de eventos e padrões em estruturas de agentes no *ProScene*.

Uma outra tecnologia utilizada é o *Processing*, uma linguagem de programação e *IDE* desenvolvida para a criação de programas visuais de múltiplos usos. Apesar de ser originalmente desenvolvida para artistas criarem artes visuais, o *Processing* é hoje uma ferramenta utilizada por diversos públicos, incluindo desenvolvedores avançados. No *ProScene* o *Processing* possui um papel fundamental de apoiar o desenvolvedor com funções prontas para cálculos e ajudar no desenvolvimento visual da simulação.

A forma como a simulação é projetada no *ProScene* faz uso de modelos baseados em agentes, da mesma maneira que o *RePast* utiliza, principalmente em sua versão *ReLogo*. O *ReLogo* utiliza o *Logo* em sua base de funcionamento, uma linguagem de programação educacional visual que possui semelhanças com o *Processing*, uma das tecnologias utilizadas pelo *ProScene*, em suas diversas funções visuais prontas para uso tais como as cores e as formas dos objetos.

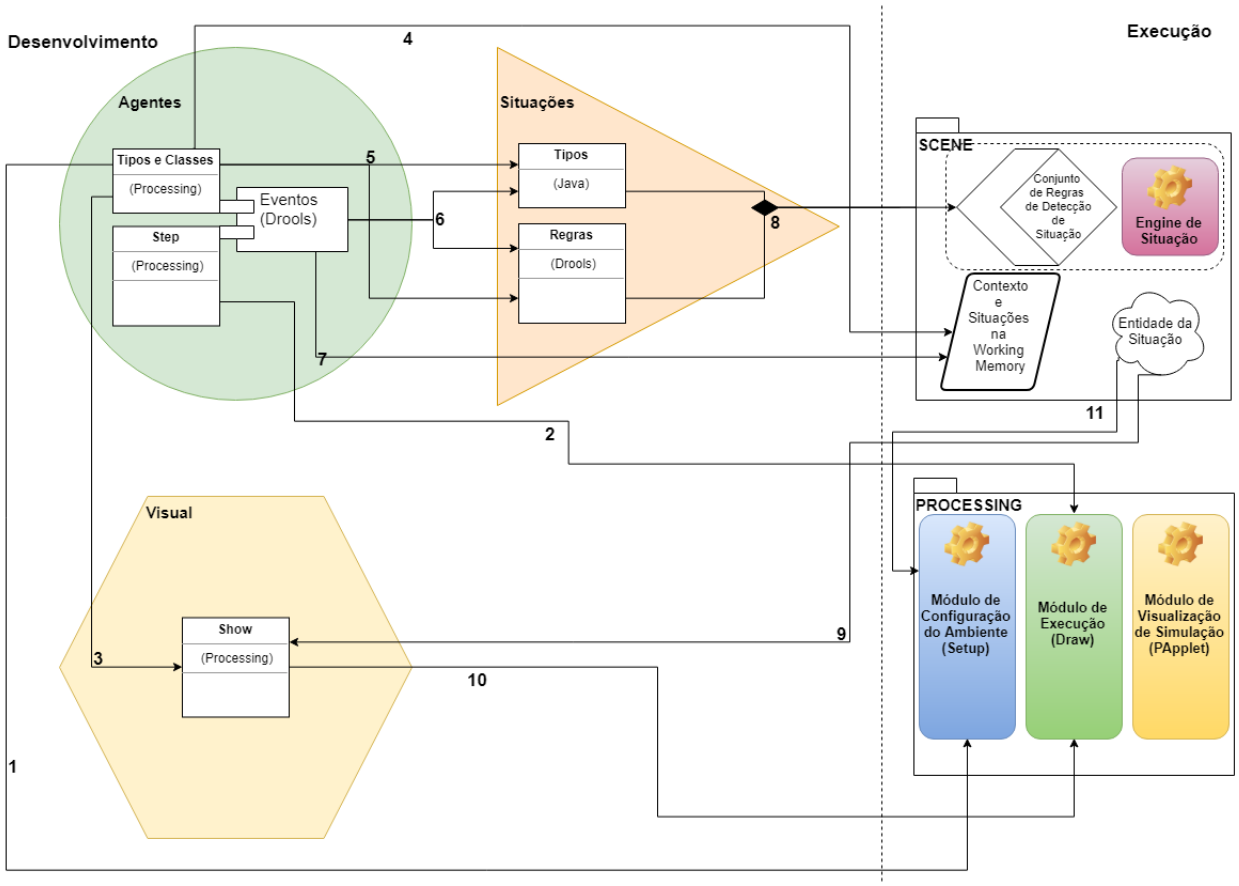

Figura 59 – Arquitetura de Implementação

Ao lado esquerdo da arquitetura de implementação (Figura 59) encontra-se os **agentes**, as **situações** e o **visual** a ser implementado pelo desenvolvedor.

Os **agentes** são os indivíduos, ou entidades, da simulação que devem ser implementados em função de sua estrutura e comportamento. Os **agentes** são elaborados no *ProScene* de forma a facilitar o desenvolvimento da estrutura de simulação e, consequentemente, a programação da simulação. A estrutura é formada pelos **tipos e classes** implementados no ***Processing***. Os **tipos e classes** dos **agentes**, então, sofrem modificações ao decorrer da simulação em vista o que está programado no método **step**, dando "vida" às simulações. Os ***steps*** contém os códigos de comportamento dos **agentes**, facilitando a definições das ações que cada **agente** deve realizar na execução da simulação. A partir dos **steps**, os **eventos** são criados para auxiliar e identificar a ocorrência de determinados padrões na simulação, cabendo ao implementador criá-los na ocorrência de eventos interessantes.

As **situações** visam o monitoramento e a identificação de determinados padrões de ocorrências que se mantém ativas. Para tanto, o programador deve implementar os tipos de situação (no ***Java***) e as regras de detecção (em *drl*, no ***Drools***

O **visual** visa a conFiguração das formas e cores que o cenário de simulação deve ter (fundo da simulação, agentes, relações, situações, entre outros). O desenvolvedor

da simulação deve utilizar o método **"show"**, implementado sob o ***Processing***, para identificar visualmente os agentes e as situações em função da localização no ambiente de simulação.

Ao lado direito da Figura 59, as máquinas de inferência de situações e simulações são encontradas, sendo o ***Scene*** e o ***Processing*** respectivamente.

Scene é o responsável por trabalhar com o **contexto e as situações na *working memory***, através de sua ***engine* de situação** faz a manutenção dos fatos e eventos na *working memory* e detecta situações com regras previamente estabelecidas. Assim, há a criação de uma nova **entidade de situação** no ***Scene*** com a detecção de situações, que representa a situação (ativa ou inativa) dado em um momento da simulação.

Processing possui a responsabilidade de conFigurar o ambiente visualmente (como tamanho do ambiente visual, cores e formas) e estruturalmente (estruturas de dados dos tipos e classes) através de seu **módulo de conFiguração do ambiente (*setup*)**. Também, as transformações visuais (mudanças de cores, formas e localização da entidades) e estruturais (transformações nos atributos dos agentes) ao decorrer da simulação são realizadas pelo módulo de execução. Para a visualização das simulações, o **módulo de visualização da simulação (*PApplet*)** exibe ao usuário a simulação como um todo.

A seguir serão apresentados os relacionamentos (arestas) da Figura 59:

1. Os tipos e classes dos agentes (definidos no *Processing*) são compilados e então executados pelo módulo de conFiguração do ambiente ("*Setup*") presente no *Processing*.

2. Os *steps*, ou o comportamento dos agentes, definido no *Processing* são atrelados aos tipos e classes e executados pelo módulo de execução ("*Draw*") do *Processing*.

3. Os tipos e classes dos agentes são utilizados pelo método *"Show"* do *processing* para a conFiguração visual de cada um, atrelando formas e cores ao agente.

4. Os tipos e classes dos agentes são inseridos na *Working Memory*, para que o *Scene* possa fazer o reconhecimento dos padrões. Com a execução da simulação, devem ser atualizados na *Working Memory*.

5. Os tipos e classes dos agentes fazem parte dos tipos e regras das situações sendo importados pelos tipos (*Java*) e utilizado na composição das regras (*Drools*).

6. Os eventos que ocorrem na simulação também fazem parte da definição dos tipos e regras das situações, podendo ser importados pelos tipos (*Java*) e utilizados na composição das regras (*Drools*).

7. Os eventos (definidos pelo *Drools*) são inseridos na *Working Memory* para utilização do *Scene* e automaticamente saem da *Working Memory* quando são expirados.

8. Os tipos (definidos em Java) e as regras (definidos em *Drools*) são inseridos no módulo do Conjunto de Regras e Detecção de Situação do *Scene*, para que a *Engine* de Situação os utilize na captação de padrões e posterior criação de uma Entidade de Situação.

9. A Entidade de Situação criada pelo *Scene* é utilizada pelo *Processing* no método "*Show*" para que as entidades sejam tratadas como agentes e exibidas na simulação quando forem situações ativas.

10. O módulo de execução ("*Draw*") faz a execução de toda a parte visual do método "*Show*". Posteriormente a parte visual é demonstrada no módulo de visualização de simulação (*PApplet*).

11. Os *steps* também são executados pelo módulo de execução do *Processing*.

Assim sendo, para a integração entre *Scene*, *Processing* e simulações orientadas a agentes (Figura 58) foi feita inicialmente com uma tentativa de inclusão do *Scene* como uma biblioteca diretamente no *Processing* através de sua *IDE* padrão. Infelizmente, o *Processing* faz uso de um sistema diferente de pastas, tendo uma incompatibilidade com a biblioteca do *Scene* em sua *IDE*.

Após a tentativa, foi feita uma mudança de abordagem: utilizando o *Processing* como uma biblioteca na *IDE* do *Netbeans* junto ao *Scene*. Assim, a biblioteca foi inclusa e houve o cuidado de executar as duas tecnologias (*Processing* e *Scene*) paralelamente. Também, o ambiente de desenvolvimento foi organizado de forma a facilitar a implementação de simulações por parte do desenvolvedor, bastando o simples download de arquivos, utilizando o conceito de simulações orientadas a agentes, fazendo a integração entre *Processing*, *Scene* e simulações baseadas em agentes. Esse desenvolvimento organizou as diferentes tecnologias para facilitar a implementação de diferentes cenários na ferramenta.

4.4 Diretrizes

Nesta Seção serão apresentadas as diretrizes para implementação de simulações e situações na plataforma *ProScene*.

4.4.1 Iniciando com *ProScene*

Conforme descrito ao decorrer da dissertação, o *ProScene* faz uso de três tecnologias principais: a linguagem / biblioteca *Processing*, a plataforma *Scene* e a noção de agentes. Dessa forma, todos os métodos e facilidades que o *Processing* possui estão disponíveis e documentados em <https://processing.org/reference/> para a utilização na visualização e no comportamento das simulações. Também, a noção de utilização da plataforma do *Scene*

pode ser encontrada em <https://github.com/pextralabs/scene-platform>. Os agentes são aqui utilizados da mesma forma que o *ReLogo* (<https://repast.github.io/index.html>), em que cada agente deve conter o seu *step* (comportamento) programado individualmente.

4.4.2 Preparação do Ambiente

Uma vez escolhido um dos ambientes de desenvolvimento para a linguagem *Java*, é necessário entender que o *ProScene* possui um *sketch*, ou seja, um rascunho de projeto em branco que possui exemplos de criação de simulações e está organizado de forma a facilitar a implementação de situações e simulações por parte do programador. Para iniciar, é preciso primeiramente fazer o download dos arquivos a partir do repositório <https://github.com/alexnede/ProScene>.

O *Sketch* deve ser importado como um projeto para a *IDE* dando início ao processo de criação das situações e simulações.

4.4.3 Definindo os Agentes da Simulação

Os agentes da simulação são todas as entidades que, de algum modo, interagem entre si ou entre o cenário. No exemplo do *Aedes aegypti* temos o mosquito, os ovos, o agente de saúde e as casas como os agentes da simulação.

```
import processing.core.*;

public class agentExample extends PApplet {

    float X;

    float Y;

    int steps = 0;

    public float getX() {
        return X;
    }

    public void setX(float X) {
        this.X = X;
    }

    public float getY() {
        return Y;
    }

    public void setY(float Y) {
        this.Y = Y;
    }

    public int getSteps() {
        return steps;
    }

    public void setSteps(int steps) {
        this.steps = steps;
    }

    public void step() {

    }
}
```

Figura 60 – Exemplo de Agente no *ProScene*.

Cada agente deve obrigatoriamente importar a biblioteca "Core" do *Processing*, extender a *PApplet* e possuir três atributos com seus respectivos *getters* and *setters*: "X", "Y" e "days" (em caso de ser um agente vivo). O exemplo da Figura 60 ilustra parte do código.

Os *floats* "X" e "Y" representam a atual localização em *pixels* no cenário do *Processing*. O inteiro "*steps*" indica o tempo de vida do agente em *steps*. Os atributos podem ser modificados de acordo com a necessidade do programador, inclusive, adicionando novos atributos. No entanto, os atributos apresentados são recomendados como atributos padrão da plataforma *ProScene* uma vez que ela se utiliza da posição dos objetos em relação ao plano para se realizar a simulação.

4.4.4 Definindo o Comportamento dos Agentes

Os agentes possuem um comportamento definido através de *steps*, ou seja, passos que deverão seguir ao decorrer da simulação. Cada *step* é definido no método público "*step*", conforme a Figura 60

No método, é possível aproveitar todos os recursos do *Processing*, sendo possível trabalhar com os dados dos agentes, cálculos matemáticos e mudanças visuais na simulação com a facilidade que o *Processing* proporciona. Um exemplo do que pode ser feito no *step* é a mudança de posição do agente (atributos "X " e "Y").

```
void simulationExecution() {
    println(tick);

    if (tick == 0) {
        updateScreen();
        //methods to show agents initially

        //example:
        /*
        simulationShow(agentss);
         */
        nextTick();
    } else {
        updateWorkingMemory();
        //methods to step agents

        //example:
        /*
        simulationStep(agentss);
         */
        updateScreen();
        //methods to show agents

        //example:
        /*
        simulationShow(agentss);
         */
        nextTick();
    }
    try {
        Thread.sleep(1000); //wait 1 second between ticks
    } catch (InterruptedException e) {
    }
```

Figura 61 – Função simulationExecution.

Após a definição do *step* dentro do agente, o programador deve atribuir um método na função "*simulationExecution*" do *sketch* que percorra os agentes e faça as ações dos agentes em cada *step*, conforme a Figura 61.

```
void simulationStep(ArrayList<Agents> elements) {
    for (int i = 0; i < elements.size(); i++) {
        Agents element = elements.get(i);
        element.step();
    }
}
```

Figura 62 – Método "*Step*" Proposto.

O método proposto como exemplo comentado no *sketch* ("*simulationStep*") pode ser visto mais claramente na Figura 62.

O método percorre toda a lista de agentes, fazendo com que cada um execute o seu *step*. Cada ação dos agentes é individualmente executada, uma por uma, de forma a não existir ações de agentes com conflito entre si.

4.4.5 Definindo o Visual dos Agentes

```
void simulationShow(ArrayList<Agents> elements) {
    for (int i = 0; i < elements.size(); i++) {
        Agents element = elements.get(i);
        imageMode(CENTER);
        image(agentsImage, element.getXcoordinate(), element.getYcoordinate(), 10, 10);
    }
}
```

Figura 63 – Método "*Show*" Proposto.

Os agentes que aparecem na visualização de simulação do *ProScene* podem apresentar-se com diferentes formas, cores, imagens ou até mesmo serem invisíveis na simulação. Para tanto, o programador deve criar um método na "*simulationExecution*" que faça a exibição dos agentes conforme a preferência pessoal. Um exemplo contido no *sketch* é o da Figura 63. O método utiliza a linguagem do *Processing* para a execução.

No exemplo, cada agente da lista é atribuído a uma imagem, criada a partir do centro com a função "*imageMode(CENTER)*", em cada "*X*" e "*Y*" do agente com um tamanho de 10 em X e 10 em Y com a função"*image(agentsImage, element.getXcoordinate(), element.getYcoordinate(), 10, 10)*".

A função "*simulationExecution*" (Figura 61) também deve ser modificada para que, assim como o step, faça a exibição de cada agente.

4.4.6 Definindo Eventos

Durante os *steps* dos agentes, determinados eventos interessantes ocorrem. Dessa forma, deve-se criar um evento e inserí-lo na *working memory* durante sua ocorrência. Cada evento possui importações de biblioteca para o *Drools*, o tempo de expiração (em segundos, minutos, dias, ...) e o(s) respectivo(s) agente(s) que faz(em) parte do acontecimento além dos métodos *getters* and *setters*.

```java
import org.kie.api.definition.type.Expires;
import org.kie.api.definition.type.Role;

@Role(Role.Type.EVENT)
@Expires("3d")

public class eventExample {

    public Agent nameagent;

    public Agent getnameagent() {
        return nameagent;
    }

    public void setnameagent(Agent nameagent) {
        this.nameagent = nameagent;
    }
}
```

Figura 64 – Exemplo de Evento.

Na Figura 64 encontra-se o código exemplo de um evento.

Os eventos podem ser atrelados a um ou mais agentes. No caso do exemplo da Figura 64, pode-se observar que há um agente atrelado "nameagent" e que tal evento tem a duração (expira) em três dias.

4.4.7 Definindo o Visual dos Eventos

Assim como os agentes, opcionalmente, os eventos também podem conter estruturas visuais através de um método próprio, como executado nas Figuras 63 e 61, em que há o *setup* do método e a inclusão do método na execução da simulação respectivamente.

4.4.8 Definindo Situações

As situações são definidas utilizando o *Drools Rule language* para a especificação das regras que fazem o casamento com os padrões. O *Scene*, por funcionar sob o *Drools*, verifica se os padrões estão sendo casados na *Working Memory* e, enquanto são casados, mantém uma situação ativa para consulta.

Dessa forma, deve-se realizar a especificação primeiramente declarando a situação e quais os agentes fariam parte da mesma.

```
declare exampleSituation  extends Situation
        agent: Agent @part
end
```

Figura 65 – Declaração de Situação.

A Figura 65 representa a declaração de uma situação "*exampleSituation*" na qual um agente é parte da situação ocorrida.

```
rule "exampleSituation"
@role(situation)
@type (exampleSituation)
    when
        agent: Agent()
        @situation: Number(this>3) from accumulate
                    ($sit: exampleEvent
                    (agent==agent), count($sit))
        then
            SituationHelper.situationDetected(drools);
end
```

Figura 66 – Exemplo de Situação.

Após a declaração, deve-se fazer a composição do padrão que casará com a situação e utilizar a função "*SituationHelper.situationDetected(drools)*" dentro do *.drl* para indicar ao *Scene* o momento de ativação ou desativação da Regra. A Figura 66 é um exemplo de situação especificada no *.drl* que se ativa e permanece ativa quando há três eventos (de exemplo) para um mesmo agente.

4.4.9 Definindo o Visual da Situação

Com a visualização de situações é possível verificar a ocorrência de determinadas características de acontecimentos em espaço e tempo e visualizá-los em sua localidade e momento ao decorrer da simulação. Com esse passo é possível, por exemplo, verificar uma situação de diversos *Aedes aegypti* em uma região por um determinado tempo, sendo um exemplo de situação crítica, representados com alguma cor e forma na simulação.

Para realizar a definição visual da situação, deve-se primeiramente especificar um agente que representa a situação. Dessa forma, o agente é criado toda vez que a situação se ativar e é retirado toda vez que a situação se desativar. Um exemplo de como isso pode ser feito é visto na Figura 67.

```
rule "SituationActive"
when
agent: Agent()
$sit: Situation(agent==agent, active==true)
then
    new allHousesAreaSituation(agent.getXcoordinate(),
        agent.getYcoordinate(), "A");
end

rule "SituationInactive"
when
    agent: Agent()
    $sit: Situation(active==false)
then
     new allHousesAreaSituation(agent.getXcoordinate(),
        agent.getYcoordinate(), "R");
end
```

Figura 67 – Exemplo de Consulta a Situação.

Na Figura 67 há uma consulta se a situação está ativa ou inativa. A partir disso,

cria-se um agente ou retira-se um agente da simulação. O agente está, então, diretamente envolvido com a situação. Uma vez criado o agente, deve-se seguir os passos de definição de visual do subcapítulo anterior "definindo o visual dos agentes".

4.4.10 Ajustes Finais e Simulando

Alguns ajustes devem ser feitos para que toda a simulação execute de forma correta e toda a estrutura implementada esteja disponível na *Working Memory* para consulta de situações. Para tanto, certifique-se de que:

- Todas as estruturas interessantes da simulação estejam inclusas na *working memory* com o comando "*session.insert(element);*".
- Todas as estruturas estejam atualizadas na *working memory* na função "*updateWorkingMemory()*" com o comando "*session.update(fh, element);*".
- O número máximo de *ticks* deve ser ajustado no inteiro "*totalticks*".
- As funções de exibição ("*show*") e de comportamento ("*step*") estão sendo chamadas na função "*simulationexecution*".
- As imagens estão localizadas nas pastas corretas, as cores estão sendo atribuídas às formas, não há sobreposição de nenhum objeto.
- Os "*ArrayLists*" estão declarados e foram inicializados corretamente.
- Opcionalmente, o método *updateScreen()* pode ser modificado para apresentar outras cores ou imagens como plano de fundo na função *background()*.
- A passagem de tempo (em "*draw()*") representa a passagem de tempo dos *steps*.

Após a conferência, basta clicar em executar e a simulação abrirá uma janela contendo a estrutura visual em tempo real de execução.

5 Estudos de Caso

Com o desenvolvimento da nova plataforma de simulação, dois estudos de casos foram realizados para a avaliação da nova plataforma. Os dois estudos foram feitos utilizando dois cenários diferentes: um cenário com o *Aedes aegypti* e um cenário com a simulação de trânsito. Os dois cenários, agora, foram analisados em vista da implementação, das situações detectadas e das possibilidades de apresentação dos artefatos (agentes e situações) de forma visual.

5.1 Cenário *Aedes aegypti*

A proposta de implementação da simulação do cenário do *Aedes aegypti* segue o mesmo modelo apresentado na Seção 3.10 utilizando o *Processing* como ferramenta de simulação.

Com a utilização do *ProScene*, foram adicionados o mapa no qual a simulação foi executada (região da UFES) seguindo a diretriz de implementação dos ajustes finais (Seção 4.4.10). Além de novos eventos e situações que incorporaram comportamentos mais elaborados e inteligentes à simulação seguindo as diretrizes de definição de eventos (Seção 4.4.6), definição de situações (Seção 4.4.8) e definição do visual das situações (Seção 4.4.9).

Sem o apoio da plataforma *ProScene*, com a utilização apenas do *Processing*, o implementador não poderia trabalhar eventos e situações na plataforma, impossibilitando a realização de análises a partir de situações. Também, a utilização do *sketch* proposto pela plataforma auxilia a implementação de simulações baseadas em agentes, com a organização do comportamento (em razão dos *steps*) e do visual dos agentes. Assim, sem o *ProScene*, o implementador teria que realizar a implementação de uma *engine* de situações e realizar manualmente a implementação de métodos e funções no *Processing* para a elaboração de simulações orientadas a agentes.

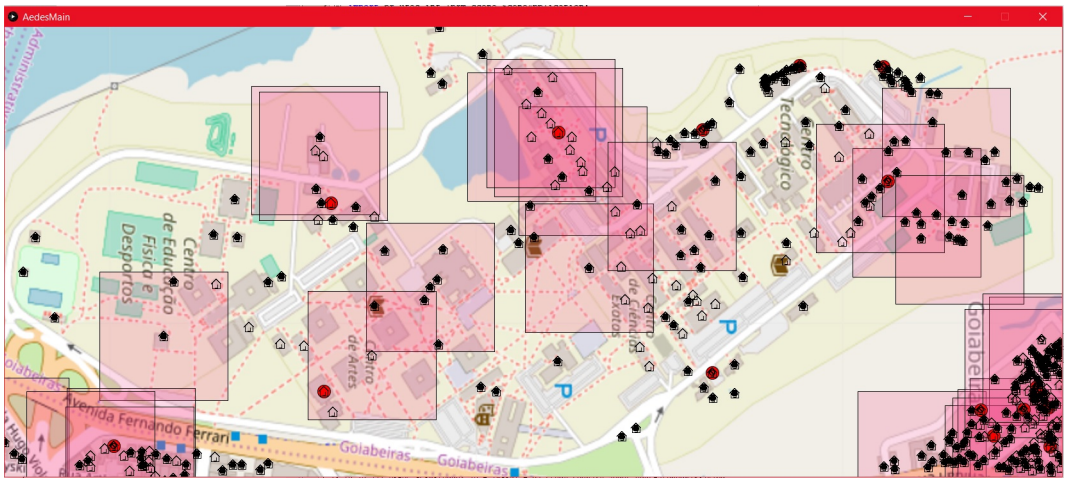

Figura 68 – Execução do Cenário de *Aedes aegypti* no *ProScene*.

Na Figura 68 é possível visualizar a simulação do *Aedes aegypti* sendo executada no *ProScene*, com situações visualmente ativas em quadrados avermelhados e bolinhas avermelhadas nas casas.

A situação "allHouseAreaMosquitoSituation" é representada pelos quadrados avermelhados, com uma situação que está ativa nas regiões que apresentam ao menos um mosquito em um raio de casas. A especificação dessa situação é ilustrada na imagem 70.

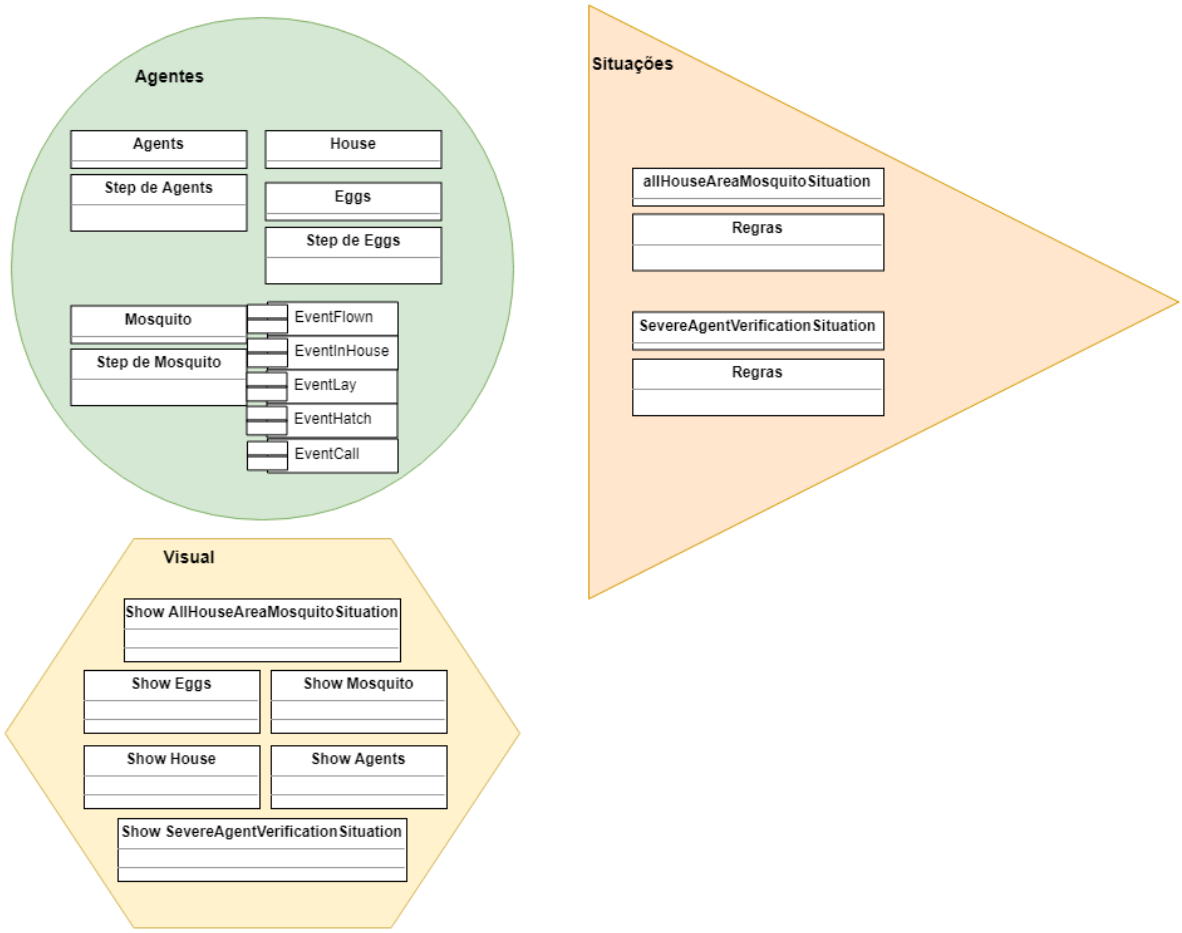

Figura 69 – Arquitetura de Implementação - Cenário de *Aedes aegypti*

A Figura 69 exibe a arquitetura do estudo de caso.

Os agentes da implementação são formados pelos tipos "*Agents*", "*House*","*Eggs*" e "*Mosquito*". "*Agents*", "*Eggs*" e "*Mosquito*" possuem seu respectivo comportamento na simulação, regido por "*Steps*". "*House*" não possui um comportamento por ser um agente que permanece imóvel durante a simulação. Os eventos "*eventFlown ()*", "*eventInhouse()*", "*eventLay()*", "*eventHatch()*" e "*eventCall()*" são criados a partir do comportamento do mosquito.

As situações são formadas pelos tipos de situação "*allHouseAreaMosquitoSituation*" e "*severeAgentVerificationSituation*", cada situação possui sua regra de ativação.

O visual da simulação é formado pelos métodos de exibição dos "*Agents*", "*Eggs*", "*Mosquito*", "*House*", "*allHouseAreaMosquitoSituation*" e "*severeAgentVerificationSituation*".

A seguir serão apresentadas as regras de situação e os códigos de exibição dos agentes de situação no *ProScene*.

```
rule "allHousesAreaMosquitosSituation"
@role(situation)
@type(allHousesAreaMosquitosSituation)
    when
        house: House()
        $event: mosquitoInHouse(house==house)
        forall(
            house2: House(X!=house.getX(),Y!=house.getY(),
                X-house.getX()<200&&X-house.getX()>0)||
                (house.getX()-X<200&&house.getX()-X>0),
                (Y-house.getY()<200&&Y-house.getY()>0)||
                (house.getY()-Y<200&&house.getY()-Y>0))
            $event2: mosquitoInHouse(house==house2)
        )
    then
        SituationHelper.situationDetected(drools);
end
```

Figura 70 – Situação *"allHouseAreaMosquitoSituation"*.

Na Figura 70 há a exibição do código em *Scene* da regra de situação "*allHouseAreaMosquitoSituation*". O código verifica se há um evento do tipo "*mosquitoInHouse*", ou seja, mosquito em casa, e verifica se todas as outras casas da redondeza em um raio de 200 *pixels* do ponto de partida inicial também possuem um evento do tipo "*mosquitoInHouse*". Dessa forma, a situação se torna ativa uma vez que as condições são satisfeitas e permanece ativa enquanto a simulação estiver sendo executada e a condição permanecer satisfeita.

```
rule "allHousesAreaMosquitosSituationActive"
when
house: House()
$sit: allHousesAreaMosquitosSituation(house==house, active==true)
then
    new allHousesAreaSituation(house.getXcoordinate(), house.getYcoordinate(),
        "A");
end

rule "allHousesAreaMosquitosSituationInactive"
when
    house: House()
    $sit: allHousesAreaMosquitosSituation(active==false)
then
     new allHousesAreaSituation(house.getXcoordinate(), house.getYcoordinate(),
        "R");
end
```

Figura 71 – Criação de Agente da Situação *"allHouseAreaMosquitoSituation"*.

Uma vez que a situação se torna ativa, um agente pertencente a situação é criado (a partir do *.drl*) como mostrado na Figura 71). O código superior detecta a ativação da situação e cria o agente representando o ponto de ativação da situação, o código inferior detecta a desativação da situação e cria o agente representando o ponto de desativação da situação.

```
void simulationShowAS(ArrayList<allHousesAreaSituation> elements) {
    for (int i = 0; i < elements.size(); i++) {
        allHousesAreaSituation element = elements.get(i);
        fill(244, 66, 134, 50);
        rectMode(CENTER);
        rect(element.getXcoordinate(), element.getYcoordinate(), 200, 200);
    }
}
```

Figura 72 – Representação Visual da Situação *"allHouseAreaMosquitoSituation"*.

A representação visual da situação é formada pelo código na imagem 72. Há uma coloração inicial em "*fill*", com uma mistura de cores *RGB* e uma máscara de transparência *alfa* de 50%. Após isso há uma função para se formar um quadrado a partir do centro em "*rectMode(CENTER)*". "*Rect*" faz o elemento quadrado a partir das coordenadas "*X*" e "*Y*" de tamanho 200 em X e 200 em Y. A escolha dos elementos visuais que farão parte da interface do *Processing* é do implementador, que pode utilizar as funções prontas e documentadas do *Processing* para o desenho. Essa escolha é realizada uma vez que a situação passa a se comportar como um agente comum no *Processing*.

```
rule "severeAgentVerificationSituation"
@role(situation)
@type(severeAgentVerificationSituation )
        when
                house: House()
                $situation: Number(this>3) from accumulate
                    ($sit: capturedMosquitoAndCallingAgent
                    (houseWithMosquito==house), count($sit))
        then
                SituationHelper.situationDetected(drools);
end
```

Figura 73 – Situação "*severeAgentVerificationSituation*".

Uma outra situação é a "*severeAgentVerificationSituation*", em que há a situação do mosquito ser capturado três vezes em uma mesma armadilha. O código na Figura 73 ilustra a especificação dessa situação.

É possível observar no código que há um *accumulate* acumulando e contando o número de eventos "*CapturedMosquitoAndCallingAgent*", ou seja, eventos em que ocorrem a captura de mosquito pela armadilha e a posterior chamada do agente para se combater o mosquito. O código ativa a situação "*severeAgentVerificationSituation*" que fica ativa, criando um agente correspondente (assim como na Figura 71 e permanece ativa enquanto as condições forem satisfeitas. As situações ativas podem ser visualizadas com o círculo vermelho em torno da casa que possui a armadilha (conforme a Figura 68).

```
void simulationShowS(ArrayList<SevereSituation> elements) {
    for (int i = 0; i < elements.size(); i++) {
        SevereSituation element = elements.get(i);
        fill(255, 0, 0);
        ellipseMode(CENTER);
        ellipse(element.getXcoordinate(), element.getYcoordinate(), 20, 20);

    }
}
```

Figura 74 – Representação Visual da Situação "*severeAgentVerificationSituation*".

O código na Figura 74 demonstra a forma visual da situação no *ProScene* fazendo a coloração inicial da forma em vermelho através da função "*fill*", inicia a formação da forma pelo centro com a função "*ellipseMode(CENTER)*" e faz a colocação da forma "*ellipse*" nas coordenadas "X" e "Y" iniciais com um tamanho 20 em X e Y.

Os resultados, discussão e avaliação serão mostrados na seção 5.3 em conjunto ao próximo cenário a ser apresentado.

5.2 Cenário Trânsito

Um dos maiores problemas nas grandes cidades é o controle do fluxo de tráfego de veículos, que cada vez mais está crescendo. Especificamente na região metropolitana de Vitória (Espírito Santo - Brasil) encontra-se a *Ponte Deputado Darcy Castello de Mendonça*, também conhecida como Terceira Ponte, sendo um dos caminhos que fazem a ligação entre a cidade de Vitória e Vila Velha. A Terceira Ponte é administrada pela Rodosol, uma empresa que possui a concessão do governo estadual para administrar, cuidar e melhorar a Terceira Ponte. Com o fluxo cada vez maior de carros nessa região, uma das alternativas introduzidas pela Rodosol para reduzir o congestionamento é a cobrança de pedágio em apenas um dos lados do fluxo de trânsito, como ocorre na Ponte Rio-Niterói. Nesse sentido, essa dissertação modela esse cenário e verifica situações que possam ser interessantes para a investigação dos acontecimentos de congestionamento no trânsito utilizando a plataforma *ProScene*.

Esse cenário possui comportamentos de interesse uma vez que possa congestionar uma das faixas da ponte. Pretende-se verificar com esse cenário:

1. O congestionamento gerado pelo pedágio e pelo afunilamento das faixas de entrada da ponte (situação em que os carros param por muito tempo).

2. O alto fluxo de veículos ao passar o pedágio (situação em que há um grande fluxo de veículos trafegando em um determinado momento).

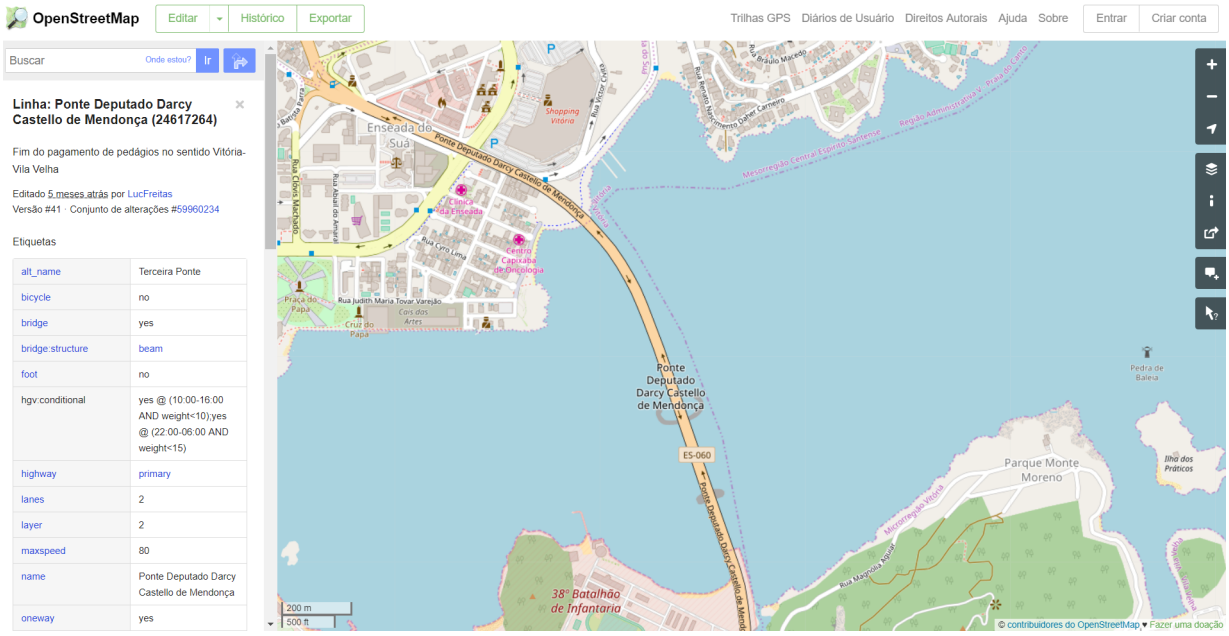

Figura 75 – *OpenStreetMap* - Terceira Ponte.

A implementação do cenário utiliza dados fornecidos pelo serviço colaborativo de mapas *OpenStreetMap* (Figura 75) e conhecimentos do usuário para a modelagem e simulação de trânsito na Terceira Ponte. Foram levados em conta os dados de localização, tamanho e velocidade da via na ponte.

A partir dos dados, foi feita uma equivalência para Agentes no *ProScene* em pixels. Lansdowne (2006), em seu artigo, explica a categoria de equivalência entre as simulações. A categoria na qual a simulação aqui apresentada se encaixa é na macroscópica, em que cada veículo é tratado da mesma forma, com velocidades e densidade iguais.

A outra categoria, microscópica, envolve a modelagem de cada indivíduo motorista, considerando fatores pessoais como o modo de direção (LJUBOVIć, 2009). Apesar da categoria microscópica envolver diversos fatores interessantes para a simulação, a macroscópica se encaixa melhor na simulação da terceira ponte em razão do cenário não possuir cruzamentos ou semáforos.

Assim, na simulação, os agentes presentes são os carros, pedágio e rodovia. A simulação foi feita de acordo com uma localização X e Y em *pixels*, ao invés de coordenadas de latitude e longitude. Essa definição em razão de *pixels* é necessária em razão do *Processing* utilizar a área visual (em *pixels*) para o desenho e a localização no ambiente de simulação.

5.2.1 Visão Geral

A simulação de trânsito na Terceira Ponte envolve os agentes da via, do carro e do pedágio.

O agente da via, simulando a rodovia da ponte, contém os trechos de rodovia (trecho inicial e trecho final) que indicam os caminhos que os carros devem seguir. Também, cada trecho contém a velocidade na qual o carro deve passar. Em (LJUBOVIć, 2009), para modelagem da via, as vias também tiveram de ser convertidas em trechos.

O agente do carro, que simula os veículos que estão trafegando, contém a sua localização em relação ao cenário e o movimento atual do carro, estando sempre sobre o agente de via. O carro percorre os trechos do agente da via, seguindo de um trecho inicial a um trecho final. Caso existam dois ou mais trechos iniciais em um trecho final de via, o agente do carro faz uma escolha aleatória para seguir um dos caminhos. Ao seguir pelo trecho, o agente do carro verifica a velocidade do agente da via e realiza cálculos para se movimentar de acordo com essa velocidade.

O agente do pedágio contém os trechos da via no qual o veículo deve parar para fazer o pagamento do pedágio, passando lentamente por estes trechos. Também, no cenário, o agente do pedágio contém um sensor para verificação do número de veículos passados.

As situações de interesse da simulação são o engarrafamento local de veículos, no qual o veículo fica parado por muito tempo, a passagem de muitos veículos em um pedágio e quando há as duas situações anteriores juntas (situação de trânsito severo).

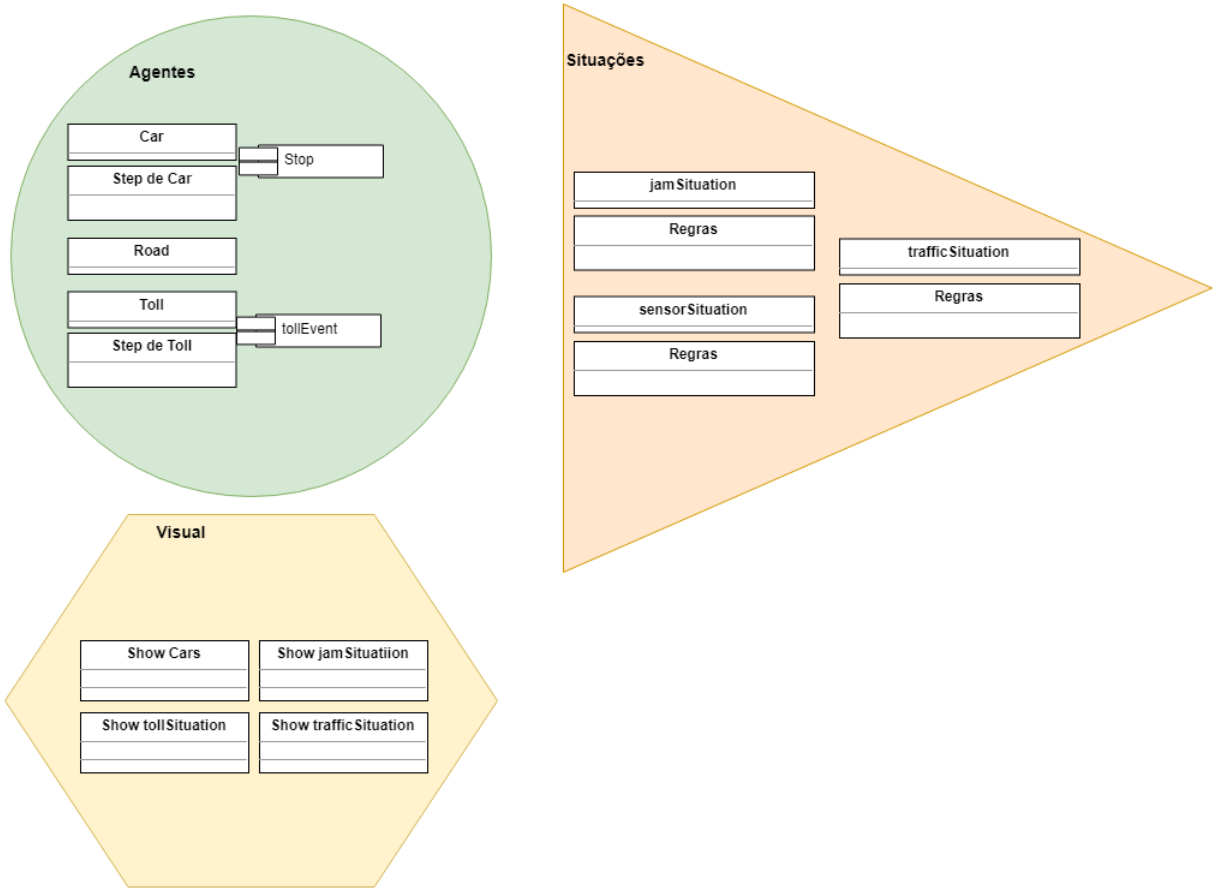

Figura 76 – Arquitetura de Implementação - Cenário de Trânsito.

A arquitetura de implementação da Figura 59 exibe a estrutura de implementação para o cenário de trânsito.

5.2.2 Criação dos Agentes da Simulação

Serão apresentadas primeiramente a estrutura dos agentes de simulação, sendo eles: a via, o carro e o pedágio.

```
import processing.core.*;

public class Road extends PApplet {

    int Xini;
    int Yini;
    int Xfin;
    int Yfin;
    int vel;

    Road(int Xini_, int Yini_, int Xfin_, int Yfin_, int vel_) {
        Xini = Xini_;
        Yini = Yini_;
        Xfin = Xfin_;
        Yfin = Yfin_;
        vel = vel_;
    }
```

Figura 77 – Agente da Via.

Os agentes que compõem a via são formados por trechos (que contém uma localização inicial e uma localização final) e velocidades em cada trecho. Para tanto, a Figura 77 exibe o agente da via e um construtor para o agente.

Na Figura 77 pode-se observar dois inteiros com a localização X e Y inicial do trecho da via ("*Xini*" e "*Yini*"), dois inteiros com a localização X e Y final do trecho da via ("*Xfin*" e "*Yfin*") e um inteiro com a velocidade no trecho da via ("*vel*").

Os agentes formados pelo carros possuem em sua composição sua localização e o movimento realizado no momento da simulação. Assim como o agente da via, há dois inteiros (''*Xpos*'' e ''*Ypos*'') para localização do agente em relação ao cenário de simulação (em relação a localização na via). Também, há um *Double* para armazenar o movimento realizado (''*movement*'') em relação aos *steps* de simulação.

O agente de pedágio simplesmente é um agente demarcador de uma localização em que o carro necessita de parar por um determinado tempo para realizar o pagamento da passagem. Assim como os outros agentes, há a localização em X e Y ("*Xpos*" e "*Ypos*") de cada pixel representante do pedágio, para que seja entendido na simulação.

5.2.3 Definição dos Comportamentos dos Agentes da Simulação

Os agentes de Via não possuem um *step* definido nesta simulação, pois são elementos fixos em uma determinada posição do cenário e não possuem uma função específica de

interesse na simulação. No entanto, é possível incorporar novos conceitos de simulação, como novos agentes e/ou novos comportamentos (*steps*) sempre que necessário.

```
void step() {
    ArrayList<Road> possibility;
    possibility = new ArrayList<Road>();
    Road choose = Main.roads.get(0);
    for (int i = 0; i < Main.roads.size(); i++) {
        Road element = Main.roads.get(i);
        if (element.Xini == this.Xpos && element.Yini == this.Ypos) {
            possibility.add(element);
        }
    }
    if (possibility.size() > 0) {
        int escolheu = (int) (random(0, possibility.size()));
        choose = possibility.get(escolheu);
    }
    if (isToll(choose.Xini, choose.Yini) == true) {
        this.movement = this.movement + 0.002;
    } else {
        if (choose.vel == 60) {
            this.movement = this.movement + 0.016;
        }
        if (choose.vel == 80) {
            this.movement = this.movement + 0.022;
        }
        if (choose.vel == 30) {
            this.movement = this.movement + 0.008;
        }
    }
    if (this.movement > 0.020 && nocar(choose.Xfin, choose.Yfin) == true) {
        this.Xpos = choose.Xfin;
        this.Ypos = choose.Yfin;
        this.movement = 0.0;
    }
}
```

Figura 78 – Comportamento do Agente Carro.

O carro exibe um comportamento muito bem definido: seguir o traçado da rodovia (definido pelo agente da via), parar se houver algum carro à frente e parar por alguns segundos nas localizações dos pedágios. Para tanto, a Figura 78 exibe os *steps* seguidos pelo carro.

```
private boolean isToll(int Xini, int Yini) {
    boolean verify = false;
    for (int i = 0; i < Main.tolls.size(); i++) {
        if (Main.tolls.get(i).Xpos == Xini
                && Main.tolls.get(i).Ypos == Yini) {
            verify = true;
        }
    }
    return verify;
}
```

Figura 79 – Função “isTool”.

Na Figura 78 é possível observar que há uma escolha inicial entre os próximos trechos de rodovia em que o carro pode andar (“*possibility*”). Os próximos trechos são escolhidos de acordo com a posição do carro (“*Xpos*” e “*Ypos*”) e a posição inicial da via

(“*Xini*” e “*Yini*”). A partir disso, é verificado se há um pedágio na via escolhida com a função “*isToll*”, que pode ser vista na Figura 79. A partir da velocidade da via, ou do pedágio, o movimento no agente de carro é incrementado e o cálculo da velocidade do veículo é feito de acordo com a transformação do movimento real para *pixels*.

```
boolean nocar(int Xfin, int Yfin) {
    boolean verify = true;
    for (int i = 0; i < Main.cars.size(); i++) {
        if (Xfin == Main.cars.get(i).Xpos && Yfin == Main.cars.get(i).Ypos) {
            verify = false;
            Stop stopped = new Stop();
            stopped.setStopped(this);
            session.insert(stopped);
        }
    }
    return verify;
}
```

Figura 80 – Função “noCar”

Uma vez incrementado o movimento, o programa verifica se há um movimento suficiente para mudar a posição do agente para o próximo trecho de via e verifica se não há carros no próximo trecho através da função “*noCar*” (Figura 80).

Os eventos que ocorrem na simulação são criados e inseridos na *working memory* do *Scene* através dos *steps* de cada agente. Para tanto, o *step* do agente carro criou o evento de engarrafamento (Figura 78), ou seja, o evento é criado caso a movimentação do veículo não seja possível (evento “*stopped*”).

```
void step() {
    for (int i = 0; i < cars.size(); i++) {
        Car element = cars.get(i);
        if(element.getXpos()==Xpos&&element.getYpos()==Ypos)
        {
            TollEvent pass = new TollEvent();
            session.insert(pass);
        }
    }
}
```

Figura 81 – Comportamento do Agente de Pedágio.

Já os *steps* do agente pedágio (Figura 81) estão relacionados a verificação do carro na passagem do pedágio, criando um evento de passagem, indicando que um carro esteve ali.

O evento de carro parado modela o acontecimento no qual o carro para e não consegue avançar para o próximo segmento de rodovia. Ele é setado durante o comportamento do agente carro (Figura 78) na chamada de função no “*noCar*” (Figura 80). Caso exista um veículo no próximo segmento, o evento é criado indicando o veículo parado e inserindo o evento na *working memory* do *Scene*.

```
import org.kie.api.definition.type.Expires;
import org.kie.api.definition.type.Role;

@Role(Role.Type.EVENT)
@Expires("20s")
public class Stop {
    private Car stopped;
    public Car getStopped() {
        return stopped;
    }
    public void setStopped(Car stopped) {
        this.stopped = stopped;
    }
}
```

Figura 82 – Evento de Carro Parado.

A Figura 82 representa o evento criado, definindo 20 segundos para a existência do evento.

O evento de passagem de pedágio modela o acontecimento de passagem de um carro no pedágio, para conhecimento pela *working memory* do *Scene*. O evento é criado quando há a passagem de um carro pelo pedágio.

```
import org.kie.api.definition.type.Expires;
import org.kie.api.definition.type.Role;

@Role(Role.Type.EVENT)
@Expires("1m")
public class TollEvent {
}
```

Figura 83 – Evento de Passagem em Pedágio.

A Figura 83 representa o evento criado, definindo 1 minuto para a existência do evento.

As situações representam os acontecimentos de interesse que acontecem e deixam de acontecer, de acordo com o desenrolar dos fatos. Para a simulação de trânsito, a situação de interesse é o *Engarrafamento Severo*, a *Passagem Constante em Pedágio* e a *Situação Crítica de Trânsito*.

```
declare jamSituation  extends Situation
        car: Car @part
end

rule "jamSituation "
@role(situation)
@type(jamSituation )
        when
                car: Car()
                $situation: Number(this>3) from accumulate($sit:
                                Jam (stopped==car), count($sit))
        then
                SituationHelper.situationDetected(drools);
end
```

Figura 84 – Situação de Engarrafamento Severo.

O engarrafamento severo ocorre quando existem três eventos de parada para o mesmo carro. Isso significa que o carro está há muito tempo parado, gerando um grande congestionamento no trânsito. A Figura 84 mostra a declaração da situação e a regra da situação.

```
rule "tollSituation"
@role(situation)
@type(tollSituation)
        when
            $situation: Number(this>24) from accumulate($sit:
                                TollEvent(), count($sit))
         then
             SituationHelper.situationDetected(drools);
end
```

Figura 85 – Situação de Passagem Constante em Pedágio.

De acordo com dados da empresa Rodosol (administradora da Terceira Ponte), há uma passagem média de 70 mil carros por dia. Segundo os cálculos, em média 48 carros passam pela ponte por minuto, sendo 24 carros para cada uma das direções. A passagem constante em pedágio (Figura 85) ocorre quando há uma passagem pelo pedágio de mais de 24 carros em um minuto. Essa situação é percebida através da regra de acumulação de 24 eventos de passagem.

```
rule "trafficSituation"
@role(situation)
@type(trafficSituation)
when
    $sit: jamSituation(active==true)
    $sit2: tollSituation(active==true)
then
     SituationHelper.situationDetected(drools);
end
```

Figura 86 – Situação Crítica de Trânsito.

Já a situação crítica de tráfego ocorre quando há ao menos uma ativação das situações de engarrafamento severo ou passagem constante em pedágio (Figura 86).

```
rule "jamSituationActive"
when
car: Car()
$sit: jamSituation(car==car, active==true)
then
    new jamSituationVisible(car.getXpos(), car.getYpos(), "A");
end

rule "jamSituationInactive"
when
car: Car()
$sit: jamSituation(car==car, active==false)
then
     new jamSituationVisible(car.getXpos(), car.getYpos(), "R");
end
```

Figura 87 – Exemplo de Criação de Agentes.

A partir da detecção das situações, há a criação de agentes de situação. A Figura 87 exibe o exemplo de criação de agentes da situação de engarrafamento severo. Os agentes das outras duas situações são criados da mesma maneira.

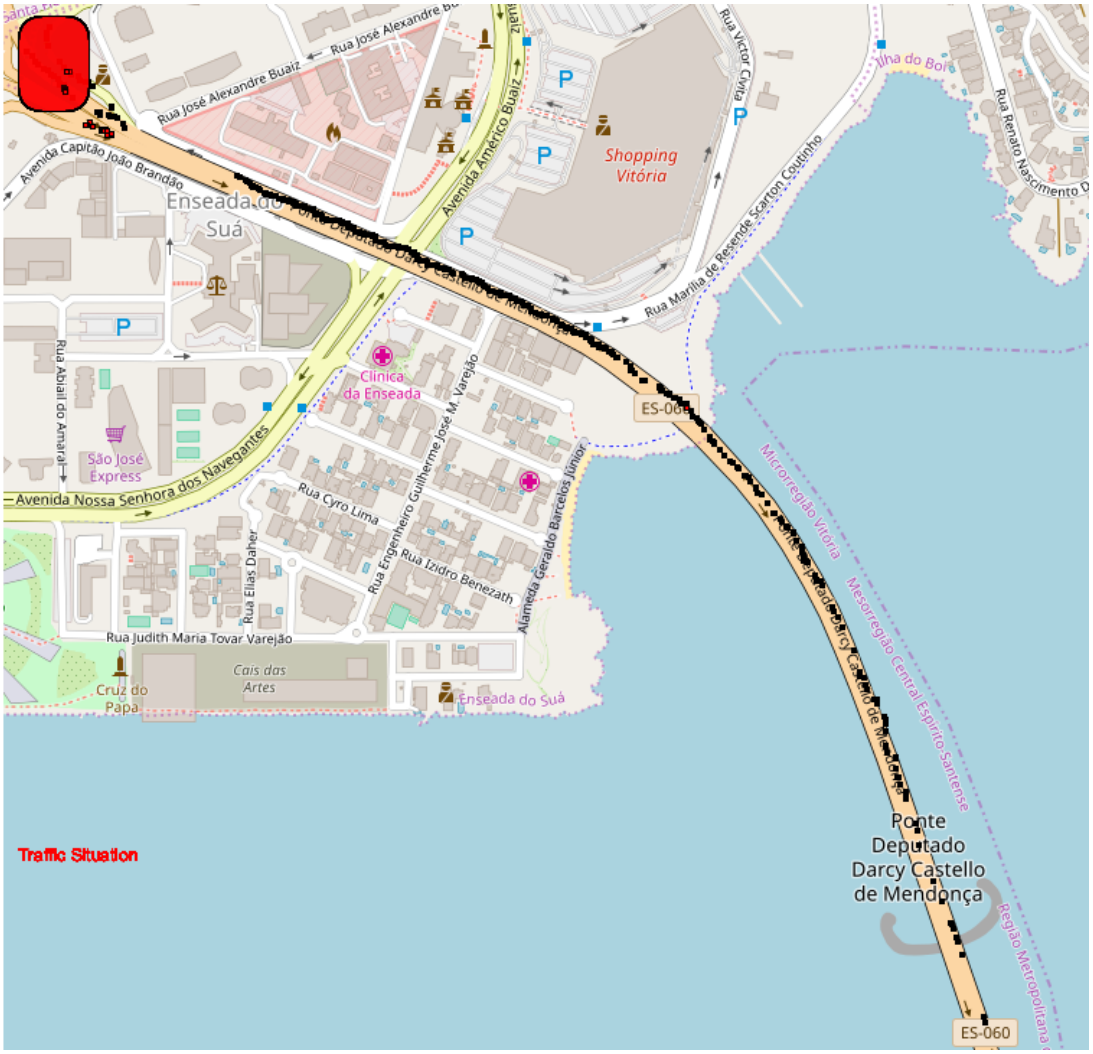

Figura 88 – Situações em Cenário de Trânsito

Assim, como visual do agente de situação de engarrafamento severo é atribuído um ponto vermelho ao carro engarrafado. Para o visual do agente da situação de passagem constante em pedágio é atribuído um quadrado vermelho sobre os pedágios. Para o agente da situação crítica de trânsito é atribuído um texto de "situação crítica de trânsito" na simulação. A Figura 88 exibe todas as situações ativas.

5.2.4 Interação com a Simulação

Esta simulação utiliza o recurso de interação com o teclado, disponibilizado por *ProScene* através das funções prontas do *Processing*.

```
public void keyPressed() {
    if (key == CODED) {
        if (keyCode == LEFT) {
            Car n = new Car(-10, -10);
            cars.add(n);
            fh=session.insert(n);
        }
        if (keyCode == RIGHT) {
            Car n = new Car(-20, -20);
            cars.add(n);
            fh=session.insert(n);
        }
    }
}
```

Figura 89 – Função "*KeyPressed*".

Para tanto, foi utilizada a função "*KeyPressed()*" do *Processing* (Figura 89), fazendo com que as ações subsequentes sejam utilizadas para a inclusão de um novo veículo de acordo com as teclas direcionais esquerda ou direita, representando os lados da via.

As vantagens da interação na criação de agentes estão na possibilidade de interação do usuário com a simulação, possibilitando a exploração de novos cenários e possibilidades ao decorrer da simulação.

5.3 Análise do ProScene

O *ProScene* é uma plataforma que integra diversas tecnologias que auxiliam na implementação de simulações em diferentes cenários. Analisando *ProScene* à luz os requisitos discutidos na Seção 4.1.1, pode-se concluir:

- A integração das duas ferramentas de implementação *Processing* e *Scene* juntamente com o conceito de simulações baseadas em agentes, proporcionou ao implementador uma plataforma única, que une o melhor de três mundos e que facilita tanto o monitoramento de situações em um ambiente simulado quanto a utilização de situações para analisar situações ocorridas em uma simulação.

- A implementação do *ProScene* possibilitou ao implementador conFigurar e demonstrar situações visualmente, no local da ocorrência da situação, oferecendo suporte particularmente à dimensão "espaço" de *Situation Awareness* (Seção 2.1). Essa visualização permite a identificação visual do momento de ativação e desativação da situação além de permitir a localização exata dos acontecimentos, melhorando o entendimento do que ocorreu no ambiente. Além disso, possíveis erros no gerenciamento do ciclo-de-vida das situações podem ser verificados visualmente durante a execução da simulação (com a avaliação do momento de ativação e desativação das situações).

- Com o uso de situações no monitoramento de ocorrências interessantes na simulação, o realismo aumentou possibilitando que o desenvolvedor faça análises mais inteligentes do ocorrido, de forma similar ao que acontece na realidade, não utilizando apenas dados estatísticos quantitativos mas também fatos interessantes no desenvolvimento da simulação.

- A utilização de cores e formas indicando uma ativação / desativação / grau de uma situação possibilita uma verificação visual de situações detectadas, indicando o momento de sua ativação e desativação e a localização de cada uma.

- A utilização de modelos orientados a agentes facilita a organização do comportamento de agentes na simulação, pois o desenvolvedor apenas se preocupa com as ações de cada agente (individualmente) que participa da simulação.

- O estudo de caso que trata do cenário de trânsito valida a proposta de ser uma ferramenta de simulação com propósito geral, isso é, o *ProScene* possibilita a simulação do *Aedes aegypti* e outros cenários (como o cenário de trânsito).

- A interação com a simulação é um recurso do *ProScene*, herdado do *Processing*, podendo o programador utilizar o mouse ou teclado para executar ações específicas durante a simulação. É possível, por exemplo, clicar em qualquer lugar da simulação para que apareçam novos agentes ou os mesmos sejam excluídos.

- O *ProScene* foca em simulações orientadas a agentes, que é o objetivo dos dois estudos de caso, porém poderia ser facilmente modificado e utilizado em outras abordagens de simulações tais como simulação baseada em eventos discretos (como proposto pela ferramenta ferramenta *SimPack* (ALLAN, 2010)) e sistemas simuladores de partículas (como proposto em (KILIAN; OCHSENDORF, 2005)) pois o *Processing* possui funções próprias para funcionamento de outros tipos de aplicações e simulações.

Os itens a seguir foram identificados como aspectos que ainda precisam ser trabalhos em *ProScene*:

- A documentação sobre *Scene* ainda é escassa até o momento da escrita desta dissertação.

- Durante a implementação do *ProScene*, foi notada que a versão mais nova do *Scene* entrava em conflito com a biblioteca *Processing*. Com a importação das bibliotecas do *Scene* e do *Processing* ao mesmo tempo, não foi possível manter uma situação ativa, ou seja, a cada passagem de tempo a situação desativava na simulação. Dessa forma, *ProScene* foi elaborado com a versão anterior do *Scene*.

- Há uma falta de ferramentas que possam fazer análises estatísticas na execução da simulação, assim como outras ferramenta fazem, como por exemplo o *RePast*, que possui em seu ambiente ferramentas prontas para análise estatística.

Assim, o *ProScene* cumpriu os requisitos inicialmente propostos, no entanto, os pontos levantados podem ser utilizados para o aprimoramento futuro da ferramenta.

6 Conclusão e Trabalhos Futuros

As *SiSAs* apresentam uma forma de se programar simulações utilizando a reação de entidades a determinados comportamentos ou situações. O conceito, idealizado nesta dissertação, está presente em diversos cenários simulados e ferramentas de simulação.

Foi conduzida nessa dissertação uma pesquisa exploratória, com base no cenário do *Aedes aegypti*, para a análise de diferentes abordagens de simulação considerando características como desempenho, análises que as ferramentas possibilitam fazer em simulações, detalhes específicos das linguagens das ferramentas e apoio para o desenvolvimento de Simulações *Situation-Aware*.

Como resultado da pesquisa, foi identificado que há uma diversidade de ferramentas que se destacaram com análise, desempenho e tipo de programação balanceadas. Dentre as ferramentas estão *Processing*, *RePast* e *Scene*.

Processing apresentou-se com funções que facilitam a implementação de simulações visuais. Já *RePast* e a simulação baseada em agentes apresentou-se de forma organizada e simples na definição de ações às entidades que compõem a simulação. Também, dentre as ferramentas de simulação, a única que utilizou o conceito de situação de forma explícita foi o *Scene*, possibilitando uma melhor abstração do cenário, de maneira a observar os acontecimentos ao decorrer da simulação.

O resultado da pesquisa exploratória serviu de base para a seleção de ferramentas que pudessem integrar uma nova plataforma. Esse conjunto de ferramentas também foi selecionado em função de características complementares, isso é, que possuem uma linguagem compatível entre si, com código aberto e com análise de simulação complementar (uma visual e outra em situações).

A integração das abordagens selecionadas deu origem a uma nova plataforma de simulação denominada *ProScene*. Uma ferramenta que possui capacidade para monitorar sequências de fatos ocorridos e demonstrar visualmente ao usuário.

ProScene possibilita a implementação de simulações orientadas a agentes, permite situações no ambiente simulado, faz uso das situações para análise de acontecimentos, demonstra as situações visualmente através de cores e formas e proporciona métodos de interação em tempo real da simulação com usuário.

A dissertação também implementa dois estudos de caso em diferentes domínios que demonstram a viabilidade de *ProScene* e indicam a generalidade da plataforma com relação à variedade de cenários que é capaz de apoiar.

Desta forma, *ProScene* atendeu aos requisitos inicialmente propostos, proporcio-

nando uma plataforma única para implementação de simulações e sistemas baseados em situações.

6.1 Discussões

Essa dissertação apresentou *ProScene*, uma plataforma de simulação baseada em agentes que também envolve o conceito de sistemas baseados em situações. A seguir serão levantadas discussões a respeito da plataforma.

Apesar do *ProScene* satisfazer os requisitos identificados na Seção 1.2, ainda é uma ferramenta que está em seu estado inicial de desenvolvimento. Faltam diversas funções que facilitem e diminuam a escrita de código, isso é, funções que possam trabalhar com os atributos dos agentes (tais como movimentá-los a outro ponto, rotacioná-los em direção a algum outro agente, entre outros). Neste sentido, apesar do *Processing* (parte integrante do *ProScene*) conter funções e métodos que possam auxiliar no desenvolvimento, outras ferramentas possuem maior escopo de métodos e funções voltados ao desenvolvimento de simulações orientadas a agentes, como o próprio *RePast*.

A interação com a simulação é uma funcionalidade interessante do *ProScene*, herdada pela integração com o *Processing* em razão das diversas funções e métodos de interação que o *Processing* possui em sua linguagem de programação. A interação é vista no estudo de caso do trânsito com a utilização do teclado na criação de novos agentes. Tal interação pode ser explorada pelo programador para tornar as simulações mais intuitivas, guiando a simulação a um determinado ponto. A interação com o mouse permitiria, por exemplo, modificar um comportamento em determinada região da simulação, o que levaria a uma simulação guiada pelo usuário. Assim, o usuário poderia guiar os resultados de uma simulação para cenários difíceis de serem encontrados em execuções aleatórias.

A implementação de novos cenários de exemplo no *ProScene* também se faz interessante uma vez que poderá melhorar a aderência da ferramenta, possibilitando o uso por novos implementadores.

6.2 Trabalhos Futuros

ProScene utiliza apenas um ambiente 2D para interação entre os agentes, nesse sentido, a extensão para ambiente 3D se faz necessária em alguns cenários, como por exemplo, simulações de dinâmica física como (ZARATTI; FRATARCANGELI; IOCCHI, 2007). O *Processing*, que integra o *ProScene*, possui suporte a objetos 3D, porém, mudanças no ambiente de desenvolvimento do *ProScene* seriam necessárias para o correto funcionamento.

Outro ponto do *ProScene* é a falta de determinados métodos e funções em sua estrutura. Outras ferramentas como *RePast* possuem métodos que facilitam a programação da locomoção e interação espacial do agente no ambiente de simulação. Como uma proposta futura, a integração dos métodos ausentes no *Processing* e presentes no *RePast* seria interessante.

O *Processing* possui um ambiente de desenvolvimento próprio e *ProScene* não faz uso do ambiente de desenvolvimento *Processing*. *ProScene* utiliza o *Processing* como biblioteca em ambientes de desenvolvimento *Java*. Como o ambiente *Processing* possui uma integração muito forte com suas derivações (isso é, as derivações como *Processing JS*, *Processing Python* dentre outras são parte da ferramenta), seria interessante uma modificação em um nível mais baixo do ambiente *Processing* e do *Scene* para o funcionamento com o *ProScene*. Essa modificação no ambiente pode fazer com que *ProScene* seja introduzida e utilizada pela comunidade *Processing*.

Até este momento foram apresentados cenários com informações próximas às informações da realidade (como por exemplo o número de mosquitos no cenário da UFES). Seria interessante aplicar o *ProScene* em um projeto que faça uso de sensores (receba dados de um dispositivo embarcado, por exemplo) para a utilização de dados de sensores em tempo real pelos agentes. Uma forma de fazer isso seria a utilização do *Firmata*, um protocolo de comunicação com dispositivos embarcados que possui uma excelente documentação e integração com *Processing*.

Também, até o momento, as análises dos cenários de simulação do *ProScene* forneceram informações aos humanos. A integração com atuadores reais, isso é, dispositivos embarcados com atuadores (como servo-motores por exemplo) poderia ser uma interessante forma das situações controlarem ambientes na realidade. Um exemplo seria a liberação automática de faixas de rodovia em um dos sentidos da terceira ponte com a detecção de situações de engarrafamento. Assim como já foi citado, o *Processing* possui uma integração excelente com o *Firmata* e isso pode auxiliar neste sentido, prevendo os acontecimentos com a utilização de simulações e atuando em casos específicos.

ProScene, até o momento, forneceu cenários pequenos de simulação como exemplos. Há a intenção de se explorar cenários com maiores agentes e outros comportamentos. Na simulação de trânsito, por exemplo, a inclusão do comportamento singular dos motoristas (como batidas por exemplo) poderia tornar a ferramenta mais realista. Isso permitiria inclusive um estudo comparativo em relação a ferramentas comerciais de simulação de trânsito.

Uma das ferramentas interessantes que foram apresentadas aqui é o *Groove* (Seção 2.2.1) em razão de sua forma visual de programação. *ProScene* não utiliza essa forma, sendo *Processing* e o *Drools* (*Scene*) a linguagem do *ProScene*. Uma forma de se aproximar do *Groove* seria a incorporação de *Situation Modeling Language* (COSTA et al., 2012) ao

ProScene, fornecendo interfaces gráficas para a especificação de agentes, seus atributos e seus comportamentos (ao invés de código).

Referências

ABOWD, G. D. et al. Towards a better understanding of context and context-awareness. In: SPRINGER. *International symposium on handheld and ubiquitous computing*. [S.l.], 1999. p. 304–307. Citado na página 7.

ADAMATTI, D. F. *AFRODITE: ambiente de simulação baseado em agentes com emoções*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, 2003. Citado na página 10.

ALLAN, R. J. *Survey of agent based modelling and simulation tools*. [S.l.]: Science & Technology Facilities Council, 2010. Citado 2 vezes nas páginas 16 e 100.

ALLEN, J. F. Maintaining knowledge about temporal intervals. In: *Readings in qualitative reasoning about physical systems*. [S.l.]: Elsevier, 1990. p. 361–372. Citado na página 20.

ALMEIDA, S. J. D. et al. Multi-agent modeling and simulation of an aedes aegypti mosquito population. *Environmental modelling & software*, Elsevier, v. 25, n. 12, p. 1490–1507, 2010. Citado na página 23.

ANGELOPOULOU, A.; MYKONIATIS, K.; KARWOWSKI, W. A framework for simulation-based task analysis-the development of a universal task analysis simulation model. In: IEEE. *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2015 IEEE International Inter-Disciplinary Conference on*. [S.l.], 2015. p. 77–81. Citado na página 21.

AXELROD, R. Advancing the art of simulation in the social sciences. In: *Simulating social phenomena*. [S.l.]: Springer, 1997. p. 21–40. Citado na página 29.

BALDI, A. et al. 2+ dengue. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2015. v. 4, n. 1, p. 186. Citado 2 vezes nas páginas 15 e 23.

BALDI, A. M. Modelo educacional adaptativo orientado ao aprendizado. *Trabalho de Conclusão de Curso em Ciência da Computação - Universidade Federal de Mato Grosso do Sul*, 2016. Citado na página 15.

BALDI, A. M. et al. Situations in simulations: An initial appraisal. In: IEEE. *2018 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA)*. [S.l.], 2018. p. 90–96. Citado 2 vezes nas páginas 1 e 2.

BALDI, A. M. et al. Simulação de aplicação de armadilhas no combate ao aedes aegypti. In: *XVII Workshop de Informática Médica - São Paulo, SP*. [S.l.: s.n.], 2017. Citado 6 vezes nas páginas 2, 7, 14, 22, 23 e 41.

BELLIFEMINE, F. et al. Jade—a java agent development framework. In: *Multi-Agent Programming*. [S.l.]: Springer, 2005. p. 125–147. Citado na página 28.

BESERRA, E. B. et al. Ciclo de vida de aedes (stegomyia) aegypti (diptera, culicidae) em águas com diferentes características. *Iheringia Ser Zool*, v. 99, n. 3, p. 281–5, 2009. Citado na página 39.

BETEL, D. et al. The microrna. org resource: targets and expression. *Nucleic acids research*, Oxford University Press, v. 36, n. suppl_1, p. D149–D153, 2008. Citado na página 16.

BORDINI, R. H.; VIEIRA, R. Linguagens de programação orientadas a agentes: uma introdução baseada em agentspeak (l). *Revista de informática teórica e aplicada. Porto Alegre. Vol. 10, n. 1 (2003), p. 7-38*, 2003. Citado na página 10.

BROWNE, P. *JBoss Drools business rules*. [S.l.]: Packt Publishing Ltd, 2009. Citado na página 18.

CARNEIRO, T. et al. Terrame - a modeling environment for non-isotropic and non-homogeneous spatial dynamic models development. In: *An International Workshop Integrated assessment of the land system: The future of land use*. [S.l.: s.n.], 2004. v. 28. Citado 2 vezes nas páginas 2 e 29.

CHRIS, L. The swarm simulation system: A toolkit for building multiagent simulations. *Working Paper 96-06-042*, Santa Fe Institute, 1996. Citado na página 29.

CHWIF, L.; MEDINA, A.; SIMULATE, T. *Modelagem e simulação de eventos discretos, 4a edição: Teoria e aplicações*. [S.l.]: Elsevier Brasil, 2015. ISBN 9788535279337. Citado 2 vezes nas páginas 9 e 10.

COLLIER, N. Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, v. 36, p. 2003, 2003. Citado na página 29.

COLUCI, V. R. et al. Ilustração de incertezas em medidas utilizando experimentos de queda livre. *Rev. Bras. Ens. Fis*, v. 35, n. 2, p. 2506, 2013. Citado na página 16.

COSTA, P. D. et al. *Rule-Based Support for Situation Management*. Cham: Springer International Publishing, 2016. 341–364 p. ISBN 978-3-319-22527-2. Citado na página 18.

COSTA, P. D. et al. A model-driven approach to situations: Situation modeling and rule-based situation detection. In: IEEE. *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*. [S.l.], 2012. p. 154–163. Citado 2 vezes nas páginas 1 e 105.

DEVLIN, K. Situation theory and situation semantics. In: *Handbook of the History of Logic*. [S.l.]: Elsevier, 2006. v. 7, p. 601–664. Citado 2 vezes nas páginas 3 e 8.

DZUL-MANZANILLA, F. et al. Indoor resting behavior of aedes aegypti (diptera culicidae) in Acapulco, Mexico. *Journal of Medical Entomology*, The Oxford University Press, 2016. Citado na página 41.

ENDSLEY, M.; GARLAND, D. *Situation Awareness Analysis and Measurement*. [S.l.]: CRC Press, 2000. ISBN 9781410605306. Citado na página 1.

ENDSLEY, M. R. Toward a theory of situation awareness in dynamic systems. In: *Situational Awareness*. [S.l.]: Routledge, 2017. p. 9–42. Citado 4 vezes nas páginas 1, 2, 3 e 7.

FERREIRA, L. S. Dengueme: um framework de software para modelagem da dengue e seu vetor. 2017. Citado 2 vezes nas páginas 2 e 23.

FILHO, P. de F. *Introdução à modelagem e simulação de sistemas: com aplicações em Arena*. [S.l.]: Visual Books, 2008. ISBN 9788575022283. Citado 2 vezes nas páginas 9 e 10.

FREITAS, R. M.; EIRAS, A. E.; OLIVEIRA, R. L. Calculating the survival rate and estimated population density of gravid Aedes aegypti (Diptera, Culicidae) in Rio de Janeiro, Brazil. *Cadernos de Saúde Pública*, v. 24, p. 2747 – 2754, 12 2008. Citado na página 42.

FRY, B.; REAS, C. Processing. org. *Processing. org*, 2018. Citado 3 vezes nas páginas 15, 17 e 29.

FUJII, H.; UCHIDA, H.; YOSHIMURA, S. Agent-based simulation framework for mixed traffic of cars, pedestrians and trams. *Transportation research part C: emerging technologies*, Elsevier, v. 85, p. 234–248, 2017. Citado na página 23.

GHAMARIAN, A. H. et al. Modelling and analysis using groove. *International journal on software tools for technology transfer*, Springer, v. 14, n. 1, p. 15–40, 2012. Citado 3 vezes nas páginas 2, 28 e 45.

GOSLING, J. et al. *The Java Language Specification, Java SE 8 Edition (Java Series)*. [S.l.]: Addison-Wesley Professional, 2014. Citado na página 28.

HELSINGER, A.; THOME, M.; WRIGHT, T. Cougaar: a scalable, distributed multi-agent architecture. In: IEEE. *Systems, Man and Cybernetics, 2004 IEEE International Conference on*. [S.l.], 2004. v. 2, p. 1910–1917. Citado 2 vezes nas páginas 2 e 28.

HOEHNDORF, R. *Situoid Theory-an ontological approach to situation theory*. Tese (Doutorado) — Master's thesis, 2005. Citado na página 8.

JOHNSON, B. J.; RITCHIE, S. A.; FONSECA, D. M. The state of the art of lethal oviposition trap-based mass interventions for arboviral control. *Insects*, Multidisciplinary Digital Publishing Institute, v. 8, n. 1, p. 5, 2017. Citado 2 vezes nas páginas 39 e 40.

JUAREZ-ESPINOSA, O.; GONZALEZ, C. Situation awareness of commanders: a cognitive model. In: *conference on Behavior representation in Modeling and Simulation (BRIMS), Arlington, VA*. [S.l.: s.n.], 2004. Citado 2 vezes nas páginas 21 e 22.

KAHN, K. et al. Modelling4all: Ambiente na web 2.0 para a construção, simulação e compartilhamento de modelos baseados em agentes. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2010. v. 1, n. 1. Citado na página 29.

KILIAN, A.; OCHSENDORF, J. Particle-spring systems for structural form finding. *Journal of the international association for shell and spatial structures*, International Association for Shell and Spatial Structures (IASS), v. 46, n. 2, p. 77–84, 2005. Citado 2 vezes nas páginas 16 e 100.

KLEIN, J. Breve: a 3d environment for the simulation of decentralized systems and artificial life. In: *Proceedings of the eighth international conference on Artificial life*. [S.l.: s.n.], 2003. p. 329–334. Citado 2 vezes nas páginas 2 e 28.

KLOPFER, E.; BEGEL, A. Starlogo under the hood and in the classroom. *Kybernetes*, MCB UP Ltd, v. 32, n. 1/2, p. 15–37, 2003. Citado na página 29.

KOKAR, M. M.; MATHEUS, C. J.; BACLAWSKI, K. Ontology-based situation awareness. *Information fusion*, Elsevier, v. 10, n. 1, p. 83–98, 2009. Citado na página 8.

KOMOSIŃSKI, M.; ULATOWSKI, S. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In: SPRINGER. *European Conference on Artificial Life*. [S.l.], 1999. p. 261–265. Citado na página 28.

KORB, K.; RANDALL, M.; HENDTLASS, T. *Artificial Life: Borrowing from Biology: 4th Australian Conference, ACAL 2009, Melbourne, Australia, December 1-4, 2009, Proceedings*. [S.l.]: Springer Berlin Heidelberg, 2009. (Lecture Notes in Computer Science). ISBN 9783642104275. Citado na página 28.

LARA, J. d.; VANGHELUWE, H. Atom3: A tool for multi-formalism and meta-modelling. In: KUTSCHE, R.-D.; WEBER, H. (Ed.). *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 174–188. ISBN 978-3-540-45923-1. Citado 2 vezes nas páginas 2 e 28.

LJUBOVIć, V. Traffic simulation using agent-based models. In: *2009 XXII International Symposium on Information, Communication and Automation Technologies*. [S.l.: s.n.], 2009. p. 1–6. Citado 3 vezes nas páginas 23, 91 e 92.

LOK, C.; KIAT, N.; KOH, T. An autocidal ovitrap for the control and possible eradication of aedes aegypti. *The Southeast Asian J. Tropical Medicine and Public Health*, v. 8, n. 1, p. 56–62, 1977. Citado 2 vezes nas páginas 39 e 40.

LUKE, S. et al. Mason: A multiagent simulation environment. *Simulation*, Sage Publications Sage CA: Thousand Oaks, CA, v. 81, n. 7, p. 517–527, 2005. Citado na página 28.

MAIMUSA, H. A. et al. Age-stage, two-sex life table characteristics of aedes albopictus and aedes aegypti in Penang Island, Malaysia. *Journal of the American Mosquito Control Association*, BioOne, v. 32, n. 1, p. 1–11, 2016. Citado na página 41.

MANCUSO, V. F. et al. idsnets: An experimental platform to study situation awareness for intrusion detection analysts. In: IEEE. *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2012 IEEE International Multi-Disciplinary Conference on*. [S.l.], 2012. p. 73–79. Citado na página 21.

MORATO, V. et al. Infestation of aedes aegypti estimated by oviposition traps in brazil. *Revista de Saúde Pública*, v. 39, n. 4, p. 553–558, 2005. Citado na página 39.

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. Citado na página 14.

OPENSTREETMAP. *OpenStreetMap*. 2017. <http://www.openstreetmap.org>. Citado na página 42.

ÖZYURT, E.; DÖRING, B.; FLEMISCH, F. Simulation-based development of a cognitive assistance system for navy ships. In: IEEE. *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conference on*. [S.l.], 2013. p. 22–29. Citado 2 vezes nas páginas 21 e 22.

PAWLEWSKI, P.; DOSSOU, P.-E.; GOLINSKA, P. Using simulation based on agents (abs) and des in enterprise integration modelling concepts. In: *Trends in Practical Applications of Agents and Multiagent Systems*. [S.l.]: Springer, 2012. p. 75–83. Citado na página 28.

PEREIRA, I. S.; COSTA, P. D.; ALMEIDA, J. P. A. A rule-based platform for situation management. In: IEEE. *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conf. on*. [S.l.], 2013. p. 83–90. Citado 7 vezes nas páginas 1, 8, 18, 19, 29, 53 e 74.

PICCIN, M. P. C. *Avaliação da relação da densidade de vetores e da presença de Aedes aegypti infectados com a ocorrência de dengue na cidade de Vitória*. Dissertação (Mestrado) — Fed. Univ. of Espírito Santo, 2013. Citado na página 42.

REPENNING, A. Agentsheets: a tool for building domain-oriented visual programming environments. In: ACM. *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. [S.l.], 1993. p. 142–143. Citado 2 vezes nas páginas 2 e 28.

ROTH, M. W. *Modeling and Simulation of Everyday Things*. [S.l.]: CRC Press, 2018. Citado 2 vezes nas páginas 2 e 3.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Malaysia; Pearson Education Limited,, 2016. Citado na página 10.

SIVAGNANAME, N.; GUNASEKARAN, K. et al. Need for an efficient adult trap for the surveillance of dengue vectors. *Indian Journal of Medical Research*, Medknow Publications, v. 136, n. 5, p. 739, 2012. Citado 3 vezes nas páginas 39, 40 e 41.

TECUCI, G.; DYBALA, T. *Building intelligent agents: an apprenticeship multistrategy learning theory, methodology, tool and case studies*. [S.l.]: Morgan Kaufmann, 1998. Citado na página 10.

TISUE, S.; WILENSKY, U. Netlogo: A simple environment for modeling complexity. In: BOSTON, MA. *International conference on complex systems*. [S.l.], 2004. v. 21, p. 16–21. Citado na página 29.

VASCONCELOS, P. F. C. Doença pelo vírus zika: um novo problema emergente nas américas? *Revista Pan-Amazônica de Saúde*, Instituto Evandro Chagas/Secretaria de Vigilância em Saúde/Ministério da Saúde, v. 6, n. 2, 2015. Citado na página 39.

WADDELL, P. Urbansim: Modeling urban development for land use, transportation, and environmental planning. *Journal of the American planning association*, Taylor & Francis, v. 68, n. 3, p. 297–314, 2002. Citado na página 29.

World Health Organization. *Dengue Control*. 2017. <http://www.who.int/denguecontrol/research>. Citado 2 vezes nas páginas 40 e 41.

WRIGHT, M. C.; TAEKMAN, J. M.; ENDSLEY, M. R. Objective measures of situation awareness in a simulated medical environment. *BMJ Quality & Safety*, BMJ Publishing Group Ltd, v. 13, n. suppl 1, p. i65–i71, 2004. ISSN 1475-3898. Citado 2 vezes nas páginas 21 e 22.

YANG, Y.; LI, J.; ZHAO, Q. Study on passenger flow simulation in urban subway station based on anylogic. *Journal of Software*, Academy Publisher, v. 9, n. 1, p. 140–146, 2014. Citado 2 vezes nas páginas 2 e 28.

YAU, S. S.; LIU, J. Hierarchical situation modeling and reasoning for pervasive computing. In: IEEE. *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*. [S.l.], 2006. p. 6–pp. Citado na página 8.

YE, J.; DOBSON, S.; MCKEEVER, S. *A review of situation identification techniques in pervasive computing. Pervasive and Mobile Computing*. [S.l.]: Elsevier, 2011. Citado na página 8.

ZARATTI, M.; FRATARCANGELI, M.; IOCCHI, L. A 3d simulator of multiple legged robots based on usarsim. In: LAKEMEYER, G. et al. (Ed.). *RoboCup 2006: Robot Soccer World Cup X*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 13–24. ISBN 978-3-540-74024-7. Citado na página 104.

Anexos

ANEXO A – “Scenary” - Implementação Java

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.Random;
4
5  public class Scenary {
6
7   private List < House > scenary = new ArrayList < House > ();
8
9   Scenary(String houses, String linkedhouses, String
       houseswithtrap, String houseswithfocus, String
       houseswithmosquito) {
10   String[] totalHouses = houses.split(",");
11   String[] totalLinked = linkedhouses.split(",");
12   String[] totalTrap = houseswithtrap.split(",");
13   String[] totalFocus = houseswithfocus.split(",");
14   String[] totalMosquito = houseswithmosquito.split(",");
15
16   for (int create = 0; create < totalHouses.length; create++) {
17    House newHouse = new House(totalHouses[create]);
18    scenary.add(newHouse);
19   }
20
21   for (int create = 0; create < totalLinked.length; create++) {
22    String link = totalLinked[create];
23    String[] linkNow = link.split("-");
24    House findone = scenary.get(0);
25    House findtwo = scenary.get(0);
26    for (int find = 0; find < totalHouses.length; find++) {
27     if (scenary.get(find).getName().equals(linkNow[0])) {
28      findone = scenary.get(find);
29     }
30     if (scenary.get(find).getName().equals(linkNow[1])) {
31      findtwo = scenary.get(find);
32     }
33    }
34    findone.addNeighborhood(findtwo);
```

```
35    findtwo.addNeighborhood(findone);
36   }
37
38   for (int create = 0; create < totalTrap.length; create++) {
39    String trap = totalTrap[create];
40    for (int find = 0; find < totalHouses.length; find++) {
41     if (scenary.get(find).getName().equals(trap)) {
42      scenary.get(find).addTrap();
43     }
44    }
45   }
46
47   for (int create = 0; create < totalFocus.length; create++) {
48    String focus = totalFocus[create];
49    for (int find = 0; find < totalHouses.length; find++) {
50     if (scenary.get(find).getName().equals(focus)) {
51      scenary.get(find).addFocus();
52     }
53    }
54   }
55
56   for (int create = 0; create < totalMosquito.length - 1; create
        ++) {
57    String mosquito = totalMosquito[create];
58    for (int find = 0; find < totalHouses.length; find++) {
59     if (scenary.get(find).getName().equals(mosquito)) {
60      scenary.get(find).addMosquitoToHouse(scenary.get(find));
61     }
62    }
63   }
64  }
65
66  public List < House > getScenary() {
67   return scenary;
68  }
69
70  public void setScenary(List < House > scenary) {
71   this.scenary = scenary;
72  }
73
74  void startSimulation(int days) {
75   long start = System.currentTimeMillis();
```

```
76
77  for (int today = 0; today < days; today++) {
78   fly();
79   hatchEggs();
80   layEggs();
81   agent();
82   clearDay(today);
83
84   boolean hasMosquitos = false;
85   boolean hasEggs = false;
86   for (int verifyingHouses = 0; verifyingHouses < this.scenary.
        size(); verifyingHouses++) {
87    House inHouse = this.scenary.get(verifyingHouses);
88    List < Mosquito > mosquitosInHouse = inHouse.getMosquitos();
89    List < Eggs > eggsInHouse = inHouse.getEggs();
90
91    for (int verifyingMosquitos = mosquitosInHouse.size() - 1;
         verifyingMosquitos >= 0; verifyingMosquitos--) {
92     hasMosquitos = true;
93    }
94    for (int verifyingEggs = eggsInHouse.size() - 1;
         verifyingEggs >= 0; verifyingEggs--) {
95     hasEggs = true;
96    }
97
98   }
99   if (hasMosquitos == false && hasEggs == false) {
100   System.out.println();
101   System.out.println();
102   System.out.println();
103   System.out.println("Simulation terminated with no Eggs/
         Mosquitos in " + today + " days.");
104   today = days;
105  }
106  System.out.println(today);
107  long elapsed = System.currentTimeMillis() - start;
108  System.out.println("Time:" + elapsed);
109  verifySituation();
110 }
111
112
113
```

```
114 }
115
116 public int randomize(int min, int max) {
117  Random random = new Random();
118  return random.nextInt((max - min) + 1) + min;
119 }
120
121 public int randomized(int min, int max) {
122  return min;
123 }
124
125 private void fly() {
126
127  for (int verifyingHouses = 0; verifyingHouses < this.scenary.
        size(); verifyingHouses++) {
128   House inHouse = this.scenary.get(verifyingHouses);
129   List < Mosquito > mosquitosInHouse = inHouse.getMosquitos();
130   for (int verifyingMosquitos = mosquitosInHouse.size() - 1;
         verifyingMosquitos >= 0; verifyingMosquitos--) {
131    Mosquito mosquitoInHouse = mosquitosInHouse.get(
          verifyingMosquitos);
132    if (mosquitoInHouse.getControl() == false) {
133     mosquitoInHouse.setControl(true); //controle do mosquito
134
135     List < House > neighborHouse = inHouse.getNeighbors();
136     if (neighborHouse.size() > 0) {
137      int chooseHouse = randomize(0, neighborHouse.size() - 1);
138      House newHouse = neighborHouse.get(chooseHouse);
139
140      while (newHouse != mosquitoInHouse.getBorn() &&
           mosquitoInHouse.getBorn().getNeighbors().contains(
           newHouse) == false) {
141       chooseHouse = randomize(0, neighborHouse.size() - 1);
142       newHouse = neighborHouse.get(chooseHouse);
143      }
144
145      List < Mosquito > newMosquitoAdd = newHouse.getMosquitos();
146
147      newMosquitoAdd.add(mosquitoInHouse);
148
149      mosquitosInHouse.remove(verifyingMosquitos);
150     } else {
```

```
151         House newHouse = inHouse;
152         List < Mosquito > newMosquitoAdd = newHouse.getMosquitos();
153 
154         newMosquitoAdd.add(mosquitoInHouse);
155 
156         mosquitosInHouse.remove(verifyingMosquitos);
157        }
158       }
159      }
160     }
161    }
162 
163    private void clearDay(int today) {
164     for (int verifyingHouses = 0; verifyingHouses < this.scenary.
          size(); verifyingHouses++) {
165      House inHouse = this.scenary.get(verifyingHouses);
166 
167      if (today % 15 == 0) {
168       inHouse.setActivefocus(true);
169      }
170 
171      List < Mosquito > mosquitosInHouse = inHouse.getMosquitos();
172      for (int verifyingMosquitos = mosquitosInHouse.size() - 1;
           verifyingMosquitos >= 0; verifyingMosquitos--) {
173       Mosquito mosquitoInHouse = mosquitosInHouse.get(
            verifyingMosquitos);
174       int dayMosquito = mosquitoInHouse.getDays();
175       dayMosquito++;
176       mosquitoInHouse.setDays(dayMosquito);
177       mosquitoInHouse.setControl(false);
178       if (dayMosquito == 38) {
179        mosquitosInHouse.remove(verifyingMosquitos);
180       }
181      }
182 
183      List < Eggs > eggsInHouse = inHouse.getEggs();
184      for (int verifyingEggs = eggsInHouse.size() - 1; verifyingEggs
           >= 0; verifyingEggs--) {
185       Eggs eggInHouse = eggsInHouse.get(verifyingEggs);
186       int dayEggs = eggInHouse.getDays();
187       dayEggs++;
188       eggInHouse.setDays(dayEggs);
```

```
189    }
190   }
191  }
192
193  private void layEggs() {
194   for (int verifyingHouses = 0; verifyingHouses < this.scenary.
        size(); verifyingHouses++) {
195    House inHouse = this.scenary.get(verifyingHouses);
196    List < Mosquito > mosquitosInHouse = inHouse.getMosquitos();
197    for (int verifyingMosquitos = mosquitosInHouse.size() - 1;
         verifyingMosquitos >= 0; verifyingMosquitos--) {
198     if (inHouse.isFocus() == true && inHouse.isActivefocus() ==
          true) {
199      inHouse.addEggs();
200     }
201     if (inHouse.isTrap()) {
202      mosquitosInHouse.remove(verifyingMosquitos);
203      inHouse.newAgent();
204     }
205    }
206   }
207
208  }
209
210  private void agent() {
211   for (int verifyingHouses = 0; verifyingHouses < this.scenary.
        size(); verifyingHouses++) {
212    House inHouse = this.scenary.get(verifyingHouses);
213    List < Agents > agentsInHouse = inHouse.getAgents();
214    for (int verifyingAgents = agentsInHouse.size() - 1;
         verifyingAgents >= 0; verifyingAgents--) {
215     Agents agent = agentsInHouse.get(verifyingAgents);
216     if (agent.getControl() == 3) {
217      agentsInHouse.remove(verifyingAgents);
218      List < Eggs > eggsInHouse = inHouse.getEggs();
219      eggsInHouse.clear();
220      House houseant = inHouse;
221      for (int visited = 0; visited < 0; visited++) {
222       List < House > neighborHouse = houseant.getNeighbors();
223
224       if (neighborHouse.size() > 0) {
225        int chooseHouse = randomize(0, neighborHouse.size() - 1);
```

```
226
227         House newHouse = neighborHouse.get(chooseHouse);
228
229         eggsInHouse = newHouse.getEggs();
230         eggsInHouse.clear();
231
232         List < Mosquito > mosquitoInHouse = newHouse.getMosquitos
                ();
233         mosquitoInHouse.clear();
234         newHouse.setActivefocus(false);
235         houseant = newHouse;
236       } else {
237         visited = 5;
238       }
239     }
240
241     } else {
242       int agentControl = agent.getControl();
243       agentControl--;
244       agent.setControl(agentControl);
245       System.out.println("Agent Control changed");
246     }
247   }
248  }
249
250 }
251
252 private void hatchEggs() {
253  for (int verifyingHouses = 0; verifyingHouses < this.scenary.
        size(); verifyingHouses++) {
254   House inHouse = this.scenary.get(verifyingHouses);
255   List < Eggs > eggsInHouse = inHouse.getEggs();
256   for (int verifyingEggs = eggsInHouse.size() - 1; verifyingEggs
         >= 0; verifyingEggs--) {
257    Eggs eggInHouse = eggsInHouse.get(verifyingEggs);
258    if (eggInHouse.getDays() == 20) {
259     inHouse.addMosquitoToHouse(inHouse);
260     inHouse.addMosquitoToHouse(inHouse);
261     inHouse.addMosquitoToHouse(inHouse);
262     inHouse.addMosquitoToHouse(inHouse);
263     inHouse.addMosquitoToHouse(inHouse);
264     inHouse.addMosquitoToHouse(inHouse);
```

```
265         inHouse.addMosquitoToHouse(inHouse);
266         inHouse.addMosquitoToHouse(inHouse);
267         inHouse.addMosquitoToHouse(inHouse);
268         inHouse.addMosquitoToHouse(inHouse);
269        }
270       }
271
272      }
273     }
274
275     private void verifySituation() {
276      int totalMosquitos = 0;
277      int totalEggs = 0;
278      for (int verifyingHouses = 0; verifyingHouses < this.scenary.
           size(); verifyingHouses++) {
279       House verify = this.scenary.get(verifyingHouses);
280       int mosquitosHere = verify.getMosquitos().size();
281       int eggsHere = verify.getEggs().size();
282       totalMosquitos = totalMosquitos + mosquitosHere;
283       totalEggs = totalEggs + eggsHere;
284      }
285      System.out.println();
286      System.out.println();
287      System.out.println("Total: " + totalMosquitos + " mosquitos and
           " + totalEggs + " eggs.");
288
289     }
290
291 }
```

ANEXO B – Situações e Regras do Scene - Implementação Scene

```
declare capturedMosquitoAndCallingAgent extends Situation
house: House @part
eggs: Eggs @part
end

declare capturedMosquitoSevereSituation extends Situation
$house: House @part
end


rule "mosquitoFlying"
when
mosquito: Mosquito()
house: House() from mosquito.getHouse()
not mosquitoFlown(this.migrated == mosquito) over window: time(1
   d)
then
mosquitoFlown eventFlown = new mosquitoFlown();
eventFlown.setMigrated(mosquito);
house.changeMosquito(mosquito);
insert(eventFlown);
update(mosquito);
end

rule "mosquitoLayingEggs"
when
mosquito: Mosquito()
house: House(trap || activefocus) from mosquito.getHouse()
not mosquitoLayedEggs(this.layed == mosquito) over window: time(1
    d)
then
mosquitoLayedEggs eventLay = new mosquitoLayedEggs();
eventLay.setLayed(mosquito);
Eggs neweggs = house.addEggs();
insert(eventLay);
insert(neweggs);
```

```
35
36 eggsHatched eventHatch = new eggsHatched();
37 eventHatch.setHatched(neweggs);
38 insert(eventHatch);
39 end
40
41
42 rule "rain"
43 when
44 not Rain() over window: time(15 d)
45
46 then
47 Rain event = new Rain();
48 insert(event);
49 end
50
51 rule "raining in a house with focus"
52 when
53 Rain()
54 house: House(focus == true)
55 then
56 house.setActivefocus(true);
57
58 end
59
60 rule "eggshatching"
61 when
62 eggs: Eggs()
63 house: House(activefocus == true) from eggs.getHouse()
64 not eggsHatched(this.hatched == eggs) over window: time(20 d)
65 then
66 Mosquito newmosquito1 = house.addMosq();
67 insert(newmosquito1);
68 Mosquito newmosquito2 = house.addMosq();
69 insert(newmosquito2);
70 Mosquito newmosquito3 = house.addMosq();
71 insert(newmosquito3);
72 Mosquito newmosquito4 = house.addMosq();
73 insert(newmosquito4);
74 Mosquito newmosquito5 = house.addMosq();
75 insert(newmosquito5);
76 Mosquito newmosquito6 = house.addMosq();
```

```
77  insert(newmosquito6);
78  Mosquito newmosquito7 = house.addMosq();
79  insert(newmosquito7);
80  Mosquito newmosquito8 = house.addMosq();
81  insert(newmosquito8);
82  Mosquito newmosquito9 = house.addMosq();
83  insert(newmosquito9);
84  Mosquito newmosquito10 = house.addMosq();
85  insert(newmosquito10);
86  house.removingEggs(eggs);
87  end
88
89
90  rule "capturedMosquitoAndCallingAgent"
91  @role(situation)
92  @type(capturedMosquitoAndCallingAgent)
93  when
94  eggs: Eggs()
95  house: House(trap == true) from eggs.getHouse()
96  eggsHatched(this.hatched == eggs) over window: time(4 d)
97  then
98  SituationHelper.situationDetected(drools);
99  end
100
101
102 rule "capturedMosquitoSevereSituation"
103 @role(situation)
104 @type(capturedMosquitoSevereSituation)
105 when
106 $house: House()
107 $situation: Number(this > 3) from accumulate($sit:
       capturedMosquitoAndCallingAgent(house == $house, active == true
       ), count($sit))
108 then
109 SituationHelper.situationDetected(drools);
110 end
111
112
113 rule "mosquitoCaptured"
114 when
115 mosquito: Mosquito()
116 $house: House() from mosquito.getHouse()
```

```
117 $sit: capturedMosquitoAndCallingAgent(house == $house, active ==
       true)
118 then
119 $house.removeMosquito(mosquito);
120 retract(mosquito);
121 System.out.println("mosquito captured");
122 end
123
124
125 rule "AgentWorking"
126 when
127 $sit: capturedMosquitoAndCallingAgent(house: house, active ==
       false)
128 then
129 house.newAgent();
130 List < Eggs > eggsInHouse = house.getEggs();
131 eggsInHouse.clear();
132 if (house.getMosquitos().size() > 0) {
133  for (Mosquito m: house.getMosquitos()) {
134   house.removeMosquito(m);
135   retract(m);
136  }
137 }
138 house.setActivefocus(false);
139 House houseant = house;
140 for (int visited = 0; visited < 5; visited++) { //visit 5 houses
141  List < House > neighborHouse = houseant.getNeighbors();
142  Random random = new Random();
143  if (neighborHouse.size() > 0) {
144   int chooseHouse = random.nextInt((neighborHouse.size() - 1 - 0)
         + 1) + 0; //choose a new house to visit
145
146   House newHouse = neighborHouse.get(chooseHouse);
147   houseant.removeAgent();
148   newHouse.newAgent();
149   eggsInHouse = newHouse.getEggs();
150   eggsInHouse.clear();
151   if (house.getMosquitos().size() > 0) {
152    for (Mosquito m: newHouse.getMosquitos()) {
153     newHouse.removeMosquito(m);
154     retract(m);
155
```

```
156     }
157    }
158    newHouse.setActivefocus(false);
159    houseant = newHouse;
160    System.out.println("Visited house " + houseant + " by agent");
161   } else {
162    visited = 5;
163   }
164  }
165  houseant.removeAgent();
166
167  end
```

ANEXO C – Comportamento do Mosquito - Implementação Processing

```
void step() {
 ArrayList < House > possibility;
 possibility = new ArrayList < House > ();
 House choose = houses.get(0);
 for (int i = 0; i < houses.size(); i++) {
  House element = houses.get(i);

  if (distance(element.getXcoordinate(), element.getYcoordinate
     (), this.getXcoordinate(), this.getYcoordinate()) < 100 && (
     element.getXcoordinate() != this.getXcoordinate() || element
     .getYcoordinate() != this.getYcoordinate())) {
   possibility.add(element);
  }
 }
 if (possibility.size() > 0) {
  choose = possibility.get(int(random(0, possibility.size())));
 } else {
  for (int i = 0; i < houses.size(); i++) {
   House element = houses.get(i);
   if (distance(element.getXcoordinate(), element.getYcoordinate
      (), this.getXcoordinate(), this.getYcoordinate()) < 100) {
    choose = element;
   }
  }
 }

 if (days < 38) {
  X = choose.getXcoordinate();
  Y = choose.getYcoordinate();
  days = days + 1;
  if (choose.getActivefocus() == true) {
   eggss.add(new Eggs(choose.getXcoordinate(), choose.
      getYcoordinate()));
  }
  if (choose.getTrap() == true) {
   agentss.add(new Agents(choose.getXcoordinate(), choose.
```

```
        getYcoordinate()));
32     this.die();
33    }
34   } else {
35    this.die();
36   }
37 }
```

ANEXO D – Comportamento dos Agentes de Saúde - Implementação Processing

```
void step() {
  ArrayList < House > possibility;
  possibility = new ArrayList < House > ();
  House choose = houses.get(0);
  for (int i = 0; i < houses.size(); i++) {
   House element = houses.get(i);

   if (distance(element.getXcoordinate(), element.getYcoordinate
      (), this.getXcoordinate(), this.getYcoordinate()) < 100 && (
      element.getXcoordinate() != this.getXcoordinate() || element
      .getYcoordinate() != this.getYcoordinate())) {
    possibility.add(element);
   }
  }
  if (possibility.size() > 0) {
   choose = possibility.get(int(random(0, possibility.size())));
  } else {
   for (int i = 0; i < houses.size(); i++) {
    House element = houses.get(i);
    if (distance(element.getXcoordinate(), element.getYcoordinate
       (), this.getXcoordinate(), this.getYcoordinate()) < 100) {
     choose = element;
    }
   }
  }

  if (days < 38) {
   X = choose.getXcoordinate();
   Y = choose.getYcoordinate();
   days = days + 1;
   if (choose.getActivefocus() == true) {
    eggss.add(new Eggs(choose.getXcoordinate(), choose.
       getYcoordinate()));
   }
   if (choose.getTrap() == true) {
    agentss.add(new Agents(choose.getXcoordinate(), choose.
```

```
        getYcoordinate()));
32     this.die();
33    }
34   } else {
35    this.die();
36   }
37 }
```