UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

EXTENSÃO SWARM INTELLIGENCE PARA O SIMULADOR ROBOCUP RESCUE

ALESSANDRO ANTONINO OSTETTO

BLUMENAU
2011

2011/1-03

ALESSANDRO ANTONINO OSTETTO

EXTENSÃO SWARM INTELLIGENCE PARA O SIMULADOR ROBOCUP RESCUE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Fernando dos Santos, Mestre, Orientador

BLUMENAU
2011

2011/1-03

EXTENSÃO SWARM INTELLIGENCE PARA O SIMULADOR ROBOCUP RESCUE

Por

ALESSANDRO ANTONINO OSTETTO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Fernando dos Santos, Mestre, Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor, Especialista – FURB
Membro:	Prof. Roberto Heinzle, Mestre, Convidado – FURB

Blumenau, 27 de junho de 2011

Dedico este trabalho a todos os amigos e família, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, pelos empurrões e cobranças, sempre presente também me apoiando.

Em especial a minha namorada Isadora Prates Costa, por seu grande apoio e incentivo.

Ao meu orientador, Fernando dos Santos, por ter acreditado na conclusão deste trabalho.

RESUMO

Swarm Intelligence é uma das áreas de estudo da Inteligência Artificial, que adota conceitos baseados na forma de organização de uma colônia de insetos para definir as ações de um programa, denominado agente. Um destes conceitos é o de comunicação pelo ambiente. O *RoboCup Rescue* é um simulador de catástrofe, baseado em sistemas multiagentes, em que um time de agentes tem como objetivo salvar uma cidade que sofreu esta catástrofe. Este trabalho aplica este conceito de comunicação pelo ambiente no desenvolvimento de uma extensão para o simulador *RoboCup Rescue* com o objetivo de melhorar o desempenho dos agentes nas suas tarefas. Experimentos realizados comprovam que o desempenho dos agentes em suas tarefas é superior quando se utiliza comunicação pelo ambiente para desenvolver um time de agentes no *RoboCup Rescue*.

Palavras-chave: *Swarm intelligence*. Sistemas multiagentes. *RoboCup rescue*.

ABSTRACT

Swarm Intelligence is one of the areas of the Artificial Intelligence, that adopts concepts based in the insects colony organization to define the actions of a program, denominated agent. One of these concepts is based on the communication through the environment. The *RoboCup Rescue* is a catastrophe simulator, based in *multiagents systems*, in which an agent team that has the objective of saving a city that has suffer this catastrophe. This report applies this communication concept through the environment by developing an extension to the *RoboCup Rescue* simulator with the objective of improving its progress and the agents in its tasks. Experiments realized prove that the performance of the agent in his tasks is superior when the communication through the environment is used to develop a team of agents in the *RoboCup Rescue.*

Key-words: *Swarm intelligence*. *Multiagents system*. *RoboCup rescue*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

IA – Inteligência Artificial

IE – Inteligência de Enxames

RCR – *RoboCup Rescue*

RF – Requisito Funcional

RNF – Requisito Não Funcional

RRSL – *Robocup Rescue Simulation League*

SI – *Swarm Intelligence*

SMA – Sistemas Multiagente

UC – Casos de Uso

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A IA é uma área de estudo da computação que procura elaborar sistemas computacionais autônomos. Alguns destes sistemas, denominados agentes, são capazes de responder a estímulos vindos do ambiente em que se encontram, produzindo algum efeito neste ambiente a fim de cumprir seus objetivos, de forma semelhante a um ser vivo racional.

Os agentes estão presentes na área de estudo da IA chamada SMA. Esta área visa o estudo de sistemas onde vários agentes devem atuar em conjunto buscando atingir um objetivo. Quando este objetivo é complexo e abrangente, ele pode estar fora da capacidade de um único agente. Então a solução é a construção de um maior número de agentes, para trabalharem em conjunto e alcançarem o objetivo, sendo este o conceito de um SMA (WOOLDRIDGE, 2002, p. 3).

Atualmente existem vários simuladores usados para testar técnicas em SMA. Entre eles estão o *RoboCup Soccer* (ROBOCUP, 2010), em que o objetivo é programar agentes para que ganhem uma partida de futebol; o Robocode (LARSEN, 2010), um simulador de batalhas entre agentes e o SeSAm (SESAM, 2010) que é um simulador para modelagem e desenvolvimento genérico de agentes. Existe também o simulador RCR (ROBOCUP RESCUE, 2010), que trabalha a ideia de programação de SMA para resolver um problema de resgate de vitimas em catástrofes.

O RCR é um simulador de SMA que apresenta uma situação de catástrofe (causada por um terremoto) com várias restrições, como bloqueio de ruas, falta de água, falta de energia elétrica e com limitações quanto ao número de mensagens enviadas entre os agentes. Neste cenário, vários agentes devem trabalhar em conjunto para efetuar, com a maior eficiência possível, o resgate das vítimas deste desastre. Cada classe de agente (como exemplo cita-se: bombeiros e policiais) tem sua função e sua inteligência é programável. Sua forma de comunicação atualmente é limitada a troca de mensagens.

SI é uma abordagem que descreve um comportamento de integração coletivo-cooperativa entre os agentes de um SMA inspirado no comportamento das colônias de insetos. Insetos são criaturas simples, com pouca capacidade de comunicação direta entre si. Para contornar esta limitação e atingir o comportamento integrado da colônia, os insetos utilizam comunicação indireta, que ocorre através de feromônios depositados no ambiente e detectados pelos outros insetos. Um exemplo é o das formigas onde uma operária marca com feromônios o caminho para o alimento encontrado, mostrando assim a localização, no

ambiente, para as companheiras. É interessante notar que, mesmo utilizando a simples comunicação indireta, a colônia de insetos é capaz de produzir um comportamento integrado bastante complexo, como a construção de ninhos (formigas) ou colméias (abelhas) (BONABEAU; THERAULAZ; DORIGO, 1999, p. 26).

Diante do exposto, este trabalho apresenta o desenvolvimento de uma extensão do simulador RCR que possibilita desenvolver um SMA que utiliza conceitos de estigmergia definidos pela SI. Para isto foi realizado um estudo aprofundado sobre a arquitetura do RCR, e alteradas as suas classes responsáveis por gerenciar os objetos do mundo, permitindo assim o depósito e leitura de feromônios. Além destas alterações, foi desenvolvido um novo componente para o RCR, responsável por gerenciar os feromônios.

Para testar a extensão foi desenvolvido um time de agentes que utilizam a comunicação pelo ambiente disponibilizada pela extensão. O desempenho deste time foi comparado com um time de agentes que não utilizam comunicação pelo ambiente. Os resultados dos experimentos demonstraram que a extensão proposta é funcional, e que o uso da comunicação pelo ambiente aumenta o desempenho dos agentes.

1.1 OBJETIVOS DO TRABALHO

O objetivo geral deste trabalho é disponibilizar uma extensão SI para o simulador RCR.

São objetivos específicos do trabalho:

a) incorporar ao RCR conceitos de SI;
b) disponibilizar uma implementação de SMA de referência para demonstrar o funcionamento da extensão e realizar testes;
c) disponibilizar um comparativo do desempenho do SMA desenvolvido com o desempenho de outros SMA´s que não utilizam a extensão.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta os assuntos relacionados ao trabalho, tais como: IA e SMA, o simulador RCR, conceitos de SI, e trabalhos correlatos. No capítulo 3 é descrito o desenvolvimento da extensão para o simulador e resultados de testes. Por fim, o capítulo 4 traz as conclusões do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 apresenta inicialmente o conceito de IA e SMA, conceitos sobre os quais o simulador RCR foi desenvolvido, assunto esse que é abrangido pela seção 2.2. Em seguida, na seção 2.3 é discutido sobre SI, assunto principal do desenvolvimento do trabalho e de onde serão utilizando os conceitos para o desenvolvimento da extensão. Por fim, a seção 2.4 apresenta alguns trabalhos correlatos.

2.1 INTELIGENCIA ARTIFICIAL E SISTEMAS MULTIAGENTE

Segundo Barr (1982, p. 3), IA é a parte da ciência da computação concentrada no desenvolvimento de sistemas computacionais inteligentes, ou seja, sistemas que exibem características que são associadas à inteligência humana tais como: entender uma linguagem, aprender, raciocinar, resolver problemas, etc.

De acordo com Wooldridge (2002, p. 15), um agente é um sistema computacional que está situado em algum ambiente e que é capaz de ações autônomas neste ambiente, a fim de cumprir os seus objetivos designados. A autonomia do agente está ligada à sua capacidade de tomar decisões em tempo de execução. O agente tem recursos necessários para perceber o ambiente e identificar a melhor alternativa a ser tomada, conseguindo assim, cumprir seu objetivo, ou função, no ambiente.

Agentes são construídos com o propósito de atingir um objetivo. Entretanto, certos objetivos podem estar além da capacidade de um único agente. Isto é o que normalmente ocorre quando o objetivo é complexo e abrangente. Nestes casos, uma das formas de atingir o objetivo é construir certo número de agentes, onde cada um deles irá atingir uma parte do objetivo geral (SYCARA, 1998, p. 79), formando um SMA.

A definição de Wooldridge (2002, p. 3), para SMA é: “Um sistema multiagente é aquele que é consistido de uma série de agentes, que interagem entre si, normalmente por troca de mensagens através de alguma infraestrutura de rede de computadores”. De acordo com Jennings, Sycara e Wooldridge (1998, p. 280) as características de um SMA são as seguintes:

- Cada agente pode possuir diferentes competências. As competências dos agentes

definem a uniformidade do SMA. Em um SMA homogêneo, todos os agentes possuem as mesmas competências. Já em um SMA heterogêneo, existem agentes com diferentes competências. Em ambos os casos, quando o objetivo estiver além das competências individuais de cada agente, eles precisarão interagir para atingi-lo;

- Cada agente pode possuir percepção limitada do ambiente. As informações necessárias para atingir o objetivo do SMA estão descentralizadas, o que limita a percepção. Isto requer interação entre os agentes para reunir as informações necessárias ao objetivo do sistema;
- Computação é assíncrona. Esta característica está ligada com a autonomia dos agentes. Por serem reativos e pró-ativos, os agentes atuam assincronamente. Se o objetivo do SMA requer algum sincronismo, os agentes devem interagir para garanti-lo;
- Inexistência de controle global/central. Não há entidade central com percepção global do sistema capaz de definir o comportamento adequado que cada agente deve realizar. Os agentes devem interagir para determinar o comportamento adequado que atinge o objetivo do SMA.

Como visto as interações entre os agentes desempenham papel fundamental em um SMA, tanto que constituem uma das principais questões estudadas na área. Jennings, Sycara e Wooldridge (1998, p. 288), classificam os tipos mais comuns de interações em:

- Cooperação. Nesta interação os agentes possuem um objetivo em comum e trabalham em conjunto para atingi-lo;
- Coordenação. Este tipo de interação tem por objetivo organizar os agentes, de forma a evitar comportamentos caóticos e prejudiciais, e explorar comportamentos organizados e benéficos;
- Negociação. Nesta interação os agentes buscam atingir um entendimento entre si.

Os agentes de um SMA podem ainda ser cooperativos ou competitivos (ou automotivados). Agentes cooperativos consideram o objetivo do SMA acima dos interesses individuais de cada agente, visando, portanto, alcançar o melhor desempenho global. Já os agentes competitivos consideram os interesses individuais acima do objetivo do sistema.

2.2 SIMULADOR RCR

Em 1995 um terremoto de grandes proporções atingiu a cidade japonesa de Kobe. Milhares de construções foram destruídas, soterrando pessoas e obstruindo ruas. Centenas de focos de incêndio surgiram e se alastraram por várias construções. A infraestrutura de energia, água e comunicação foi extremamente danificada. Aproximadamente 6.000 pessoas morreram e mais de 300.000 ficaram feridas (KITANO; TADOKORO, 2001, p. 39).

A partir desta catástrofe, notou-se a necessidade de um sistema que possa criar planos robustos, dinâmicos e inteligentes para busca e resgate que auxilie o esforço humano em situações catastróficas desta escala (KITANO; TADOKORO, 2001, p. 40). A partir disso, fundou-se a RRSL, onde o objetivo é a busca deste sistema. Esta liga disponibiliza um simulador de desastres (terremotos) e operações de resgate, chamado RCR (SKINNER; BARLEY, 2006, p. 633). Neste simulador é possível avaliar a qualidade e eficiência de abordagens multiagente no que tange ao salvamento de pessoas e minimização de danos. Questões como heterogeneidade, acesso limitado à informação, comunicação limitada e planejamento em tempo real caracterizam o RCR como um domínio multiagente complexo (KITANO; TADOKORO, 2001, p. 40).

O simulador trabalha recebendo como entrada dados geográficos (mapas, ruas, construções, etc.) e informações sobre o terremoto. Com estas informações o simulador constrói um cenário imediatamente após a catástrofe acontecer, em termos de estruturas de construções, bloqueio de ruas e vítimas feridas. Após esta fase, o simulador reproduz a evolução da catástrofe ao longo do tempo. Esta evolução inclui, por exemplo, propagação de incêndios para construções inicialmente intactas e agravamentos no estado de saúde dos civis.

Na fase de simulação há um contador de tempo, que é fixo, para toda simulação, e há também um *score*, que é calculado de acordo com a evolução dos agentes na execução das tarefas. Ele leva em consideração vários quesitos, como a quantidade de civis resgatados e também a quantidade de dano causado pelo fogo às construções. A fórmula utilizada para o cálculo do *score*, definida pelas regras da RRSL (2010, p. 9), é apresentada no Quadro 1.

$$\text{score} = \left(P + \frac{H}{H_{\text{inicial}}}\right) * \sqrt{\frac{B}{B_{\text{inicial}}}}$$

Sendo:

• *P*: quantidade total de agentes vivos;

• *H*: soma dos níveis de saúde dos agentes;

• $H_{inicial}$: soma dos níveis de saúde dos agentes no início da simulação;

• *B*: soma da área construída preservada;

• $B_{inicial}$: soma da área construída no início da simulação.

Quadro 1 – Fórmula de cálculo do *score*

Para lidar com o problema da catástrofe, o simulador incorpora alguns tipos de agentes, chamados agentes de regaste. Estes agentes são divididos em duas classes: agentes de campo, que percebem e atuam no ambiente; e agentes de central que possuem localização fixa e não percebem diretamente o ambiente (a percepção se resume a informações passadas pelo agente de campo). Os agentes de campo são: brigada de incêndio, agentes responsáveis pelo combate a focos de incêndio; força policial, responsável por remover escombros e bloqueios nas ruas e o time da ambulância, encarregado do resgate de soterrados e/ou feridos. Existem também os agentes fixos responsáveis por coordenar, respectivamente, seus agentes de campo: posto de bombeiros; delegacia de polícia e central de ambulâncias.

O modo mais rápido para que os agentes possam cumprir sua missão de resgate é que eles trabalhem em conjunto, coordenando-se para agir de modo cooperativo, pois a função de cada um pode depender da função do outro. Um exemplo é no caso de um agente do time da ambulância necessitar passar por uma rua com bloqueio para prestar socorro às vitimas. Para isso, ele irá precisar avisar um agente da força policial para que esse venha e libere passagem.

A interação entre os agentes, no RCR, é feita através de comunicação por voz ou por rádio. Em ambos os casos há severas restrições de quantidade e tamanho das mensagens enviadas, como limite e tamanho das mensagens em 256 bytes, somente agentes de campo do mesmo tipo podem enviar mensagens entre si e os agentes só recebem mensagens enviadas em um raio máximo de 30 metros. Além disso, outro fator que dificulta muito a comunicação é o fato da ocorrência de ruídos, fazendo com que em alguns momentos, as mensagens não sejam transmitidas corretamente.

A Figura 1 é uma imagem do simulador. O número 1, em vermelho, indica a exibição do mapa como uma visão geral do simulador, com os agentes bombeiros (pontos vermelhos),

os agentes policiais (pontos azuis), o agente de ambulância (ponto branco) e os civis (pontos verdes). Os polígonos de cor cinza representam edificações. As edificações que estão em chamas tem sua cor alterada para tons de amarelo e vermelho. Uma edificação onde as chamas foram apagadas fica com cor azul. As edificações que foram totalmente consumidas por incêndio ficam na cor preta. O número 3 mostra os agentes bombeiros desempenhando seu papel de apagar os incêndios em edificações. Junto ao número 4 é possível ver um agente policial limpando um bloqueio. No número 5 está representada a ação do agente ambulância de encontrar os civis.

O número 2 indica a representação do refúgio, que é para onde o bombeiro vai para encher seu tanque de água. Os civis também devem ir para os refúgios, seja encontrando o caminho ao andar pelo mapa, ou sendo levados pelos agentes de ambulância.

O tempo de execução, contabilizado por *time-steps*, é apontado pelo número 6. Logo ao lado direito é mostrado o *score* da simulação indicado pelo número 7. As janelas indicadas pelo número 8 mostram *logs* dos simuladores em execução, no caso o simulador de incêndio e o simulador de colapso. O *log* dos agentes em execução é indicado pelo número 9.

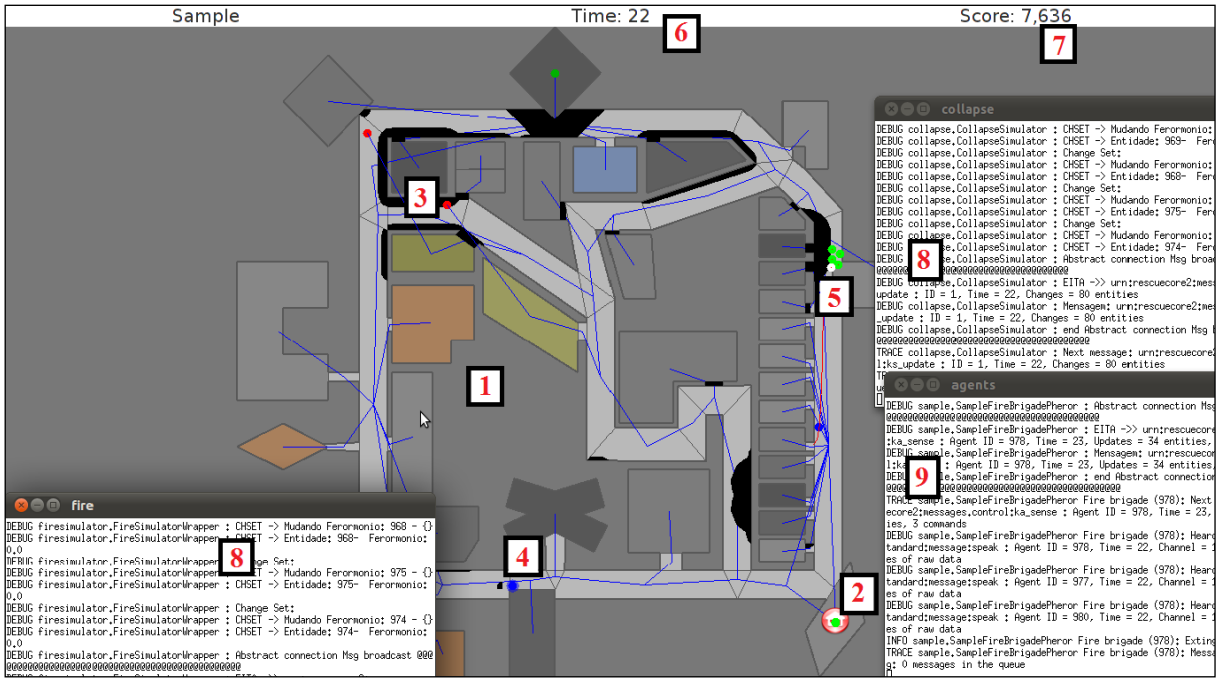


Figura 1 – Simulador RCR em tempo de execução

2.2.1 Especificação geral do simulador RCR

O simulador é desenvolvido na linguagem Java e totalmente baseado em orientação à objetos. A fim de detalhar a implementação e o funcionamento do simulador, a Figura 2 apresenta um diagrama de componentes, com os componentes que formam o RCR.

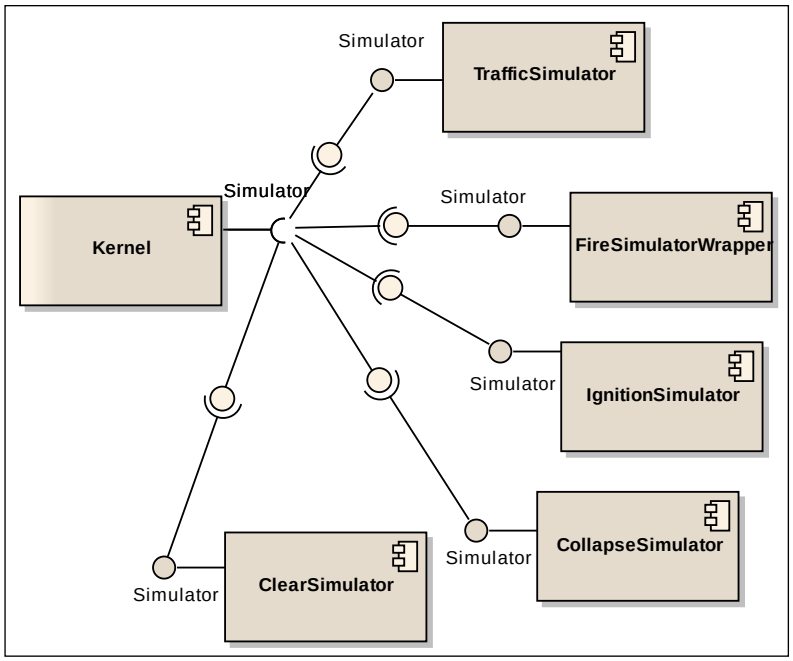


Figura 2 – Diagrama dos componentes que formam o simulador RCR

O RCR é formando por um *kernel* e por outros simuladores. O *kernel* é o componente principal do simulador responsável por gerenciar a simulação. Ele determina a percepção dos agentes, recebe os comandos dos agentes, e transmite estes comandos aos outros simuladores para que interpretem e executem o que foi requisitado. Após a execução dos simuladores, o *kernel* efetiva as alterações no mundo, atualiza o *score* e inicia um novo ciclo atualizando a percepção dos agentes.

Cada simulador tem uma função no RCR sendo responsável por interpretar comandos ligados a sua simulação. O simulador de tráfego (*TrafficSimulator*) é responsável por gerenciar a movimentação dos agentes recebendo comandos de deslocamentos com o caminho escolhido pelo agente. O *ClearSimulator* é responsável por executar os comandos de limpeza de bloqueios. Já o simulador de colapso (*CollapseSimulator*) é responsável por gerar os bloqueios e os incêndios iniciais. O *IgnitionSimulator* tem a função de gerar novos incêndios. O simulador *FireSimulatorWrapper* é responsável por gerenciar os incêndios no mapa, causando dano às edificações e também por interpretar os comandos para apagar estes incêndios.

Para que todos os componentes do RCR possam trabalhar em conjunto, o simulador utiliza-se de várias classes, com finalidade de representar os agentes, os objetos que compõem o mundo, e a execução da simulação propriamente dita. A Figura 3 apresenta um diagrama de classes simplificado do simulador RCR.

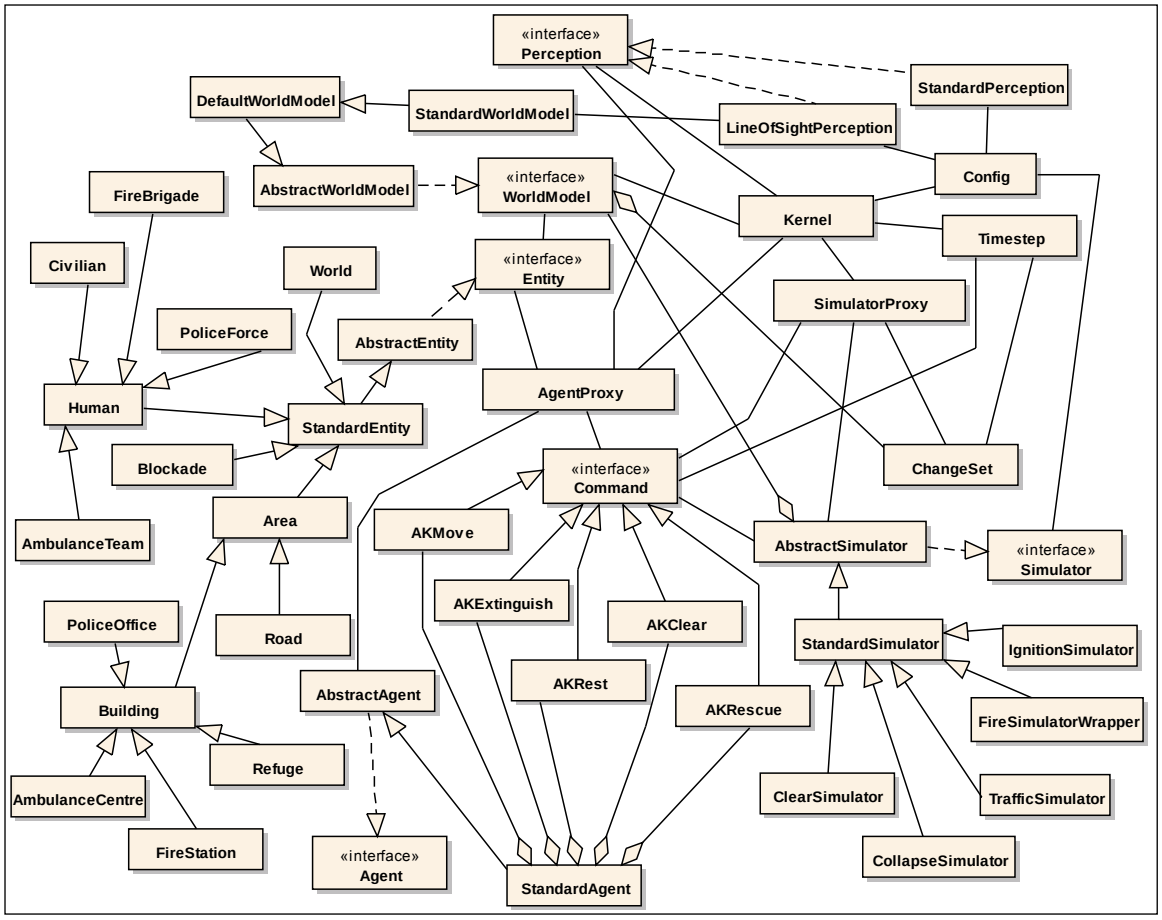


Figura 3 – Diagrama de classes simplificado do simulador

Para representar o mundo, ou seja, cenário simulador, são utilizadas várias classes. Cada objeto deste cenário é representado por uma entidade, determinada pela *interface* `Entity`, dela são realizadas todas as representações dos objetos do mundo, como os bloqueios (classe `Blockade`). As ruas são representadas pela classe `Road`, uma extensão da classe `Area`. Desta classe também é estendida a representação das construções pela classe `Building`. Também são extensões da classe `Building`: o refúgio, definido pela classe `Refuge`; as estações dos bombeiros (`FireStation`); os centros das ambulâncias (`AmbulanceCentre`) e os escritórios da policia (`PoliceOffice`. Além destas, existe uma representação do mundo no que diz respeito às condições naturais, como ventos, definida na classe `World`. Todas estas classes estendem a classe `StandardEntity` que possui a definição padrão das entidades do mundo. Esta classe é uma extensão da classe `AbstractEntity` que define a base para qualquer implementação da *interface* `Entity`.

Os agentes também são considerados entidades, assim eles possuem uma representação básica que é estendida da classe `Human`. Esta classe possui apenas os atributos comuns a todos os agentes, inclusive dos civis. Cada agente tem uma especificação de sua estrutura estendendo a classe `Human`. O agente bombeiro tem sua especificação pela classe `FireBrigade`, o agente de polícia pela classe `PoliceOfficer`, o agente ambulância pela `AmbulanceTeam` e os civis pela classe `Civillian`.

A *interface* `WorldModel` define a estrutura para o cenário. Sua implementação (`AbstractWorldModel`) tem definidos os métodos responsáveis pela atualização das propriedades das entidades do mundo. Estendendo ela, a classe `DefaultWorldModel` tem a função de gerenciar as entidades do mundo, no processo de inclusão e exclusão. Por fim, a classe `DefaultWorldModel`, que estende a classe `AbstractWorldModel` define os padrões para a localização geográfica dos objetos no mundo. É através desta classe também que os simuladores têm acesso às entidades do mundo.

Os simuladores são definidos pela *interface* `Simulator`. Esta *interface* é implementada pela `AbstractSimulator` que define os principais métodos utilizados pelos simuladores para se comunicarem com o `Kernel`. Esta classe é estendida pela `StandardSimulator` que padroniza a conexão dos simuladores com o `WorldModel`. Estendem a classe `StandardSimulator` as representações dos simuladores. A classe `FireWrapperSimulator` para o simulador de incêndio. As classe `ClearSimulator` e `CollapseSimulator` os simuladores de limpeza de bloqueios e colapso, respectivamente. A classe `TrafficSimulator` contém a definição dos simulador de tráfego.

Além de serem entidades no mundo, os agentes tem sua estrutura de conexão com o `Kernel` definida pela *interface* `Agent`. A classe `AbstractAgent` padroniza os métodos base para a interação dos agentes com o mundo e a implementação dos mesmos pelo usuário. Dentro destes métodos está inclusa a definição e atualização da percepção em relação aos objetos do mundo pelo agente. Estendendo esta classe, a `StandardAgent` define os métodos para o envio das mensagens que representam os comandos. Para a implementação de um agente o usuário deve estender a classe `StandardAgent`.

A classe `Kernel`, que representa o componente *kernel*, armazena a representação do mundo (cenário do simulador) e é responsável por gerenciar a simulação ao longo dos *time-steps*, recebendo as ações dos agentes e encaminhando estas ações para os simuladores encarregados de executá-las. Os *time-steps* são representados pela classe `Timestep`, que armazena as mudanças ocorridas naquele *time-step*. Estas mudanças são armazenas e gerenciadas pela classe `ChangeSet` que então é salva pelo `Timestep`.

Em cada *time-step*, os agentes enviam comandos ao `Kernel`, através da classe `AgentProxy`, que é responsável pela comunicação entre o `Kernel` e os agentes. É esta classe que também é responsável por repassar a percepção do mundo, atualizada pelo `Kernel`, para os agentes. A percepção é definida pela *interface* `Perception` e representa o que o agente pode percebe no mundo, tanto no quesito de objetos (ruas, construções, centros, etc.) quanto agentes (civis, policias, etc.) e suas propriedades (grau de incêndio, temperatura, bloqueios, vida, posição, etc.).

Os comandos são ações dos agentes, que são enviados ao `Kernel` de acordo com o mundo que eles perceberam naquele *time-step*. Eles são representados pela *interface* `Command` e pelas classes que realizam esta *interface*.

Os comandos principais são definidos pelas classes: `AKMove` para a locomoção do agente pelo mapa; `AKExtinguish` para apagar um incêndio; `AKClear` para a limpeza dos bloqueios das ruas; `AKRescue` para resgatar um civil; `AKRest` para quando o agente estiver no refúgio para, por exemplo, encher o tanque com água. Existem outras mensagens definidas para o uso dos agentes, mas elas não são de interesse a este trabalho.

Após o receber os comandos, o `Kernel` envia-os para os simuladores conectados a ele através da classe `SimulatorProxy`, que é responsável pela iteração entre `Kernel` e simuladores. Cada simulador recebe todos os comandos enviados e filtra se aquele comando que lhe diz respeito. A partir dos comandos recebidos, cada simulador executa o comando no mundo, fazendo as alterações requisitadas e adicionando as modificações a um `ChangeSet` associado ao `Kernel`.

Assim que todos os simuladores terminaram de processar os comandos, o `Kernel` tem preenchido no `ChangeSet` um *hash* com todas as alterações requisitadas pelos simuladores. Em seguida, o `Kernel` atualiza a representação do mundo enviando para o `WorldModel` o `ChangeSet` com as mudanças.

Por fim, a visualização do mundo (mostrada na Figura 1) é atualizada e o *score* é contabilizado de acordo com as modificações, terminando assim o *time-step*.

2.2.2 Especificação do componente *TrafficSimulator*

O componente *TrafficSimulator* é responsável por gerenciar a locomoção dos agentes pelo mapa. Esta seção descreve com mais detalhes este componente, em função de que o

mesmo será alterado neste trabalho para atingir os objetivos propostos.

A Figura 4 apresenta o diagrama de classes com as classes envolvidas no componente *TrafficSimulator*, e sua relação com o *kernel* e agentes.

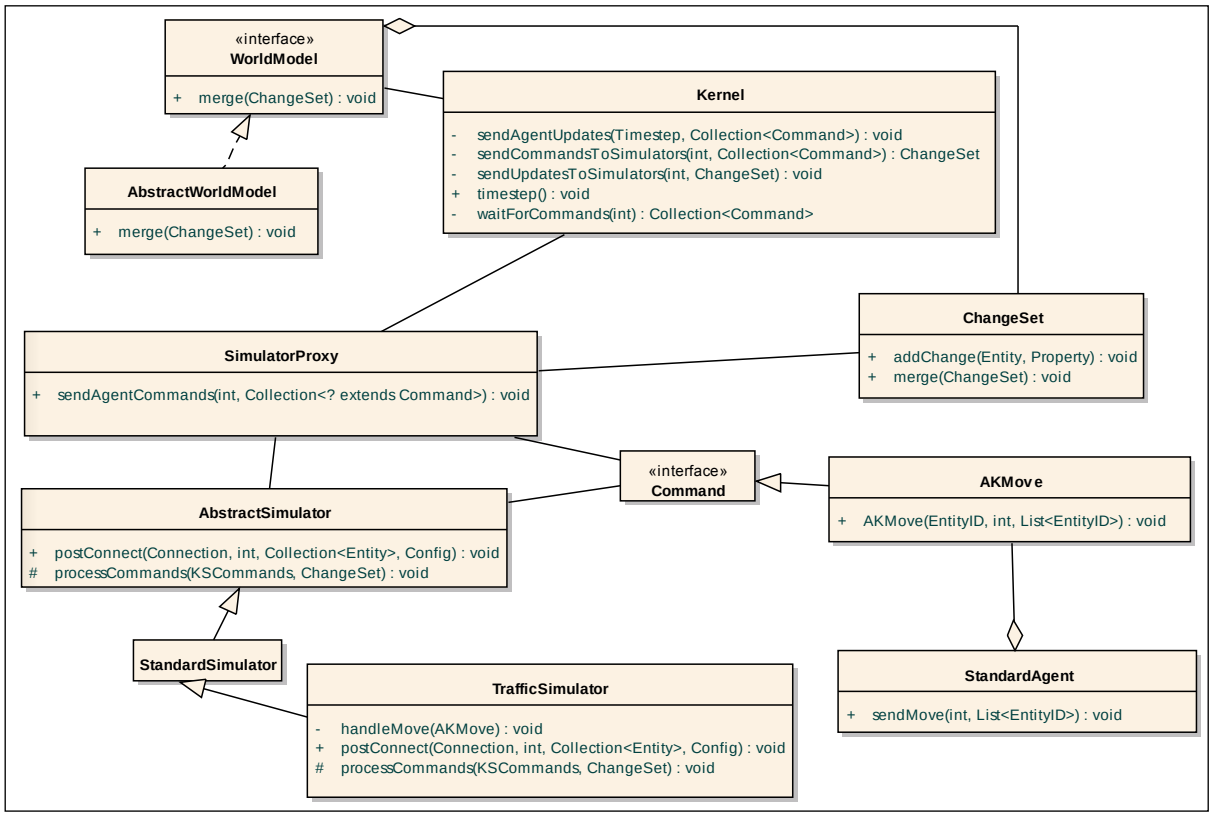


Figura 4 – Especificação detalhada do simulador de tráfego e sua relação com a classe `Kernel`

Cada agente que deseja se locomover no mundo deve "montar" um caminho a ser seguido. Este caminho é composto pelos identificadores das entidades por onde o agente vai passar (como ruas, construções), é enviado para o `Kernel` através do método `sendMove`. O método `sendMove`, por sua vez usa um comando `AKMove` para enviar o caminho ao `Kernel`.

O `kernel` envia estes comandos para a classe `SimulatorProxy` através do método `sendCommandsToSimulators`, que repassa os comandos para a classe `AbstractSimulator` utilizando o método `sendAgentsCommands`. Então a classe `AbstractSimulator` envia os comandos para o simulador `TrafficSimulator` através do método `processCommands`. Ao receber os comandos do *kernel*, o simulador reconhece o comando de movimentação e o interpreta, chamando assim seu método privado `handleMove`. No `handleMove` é definido o deslocamento do agente com a distância que ele irá percorrer. Por fim, o `TrafficSimulator` adiciona essa modificação ao `ChangeSet` do `Kernel` com o método `addChange`.

Após a execução de todos os simuladores, a classe `SimulatorProxy` unifica as modificações no `ChangeSet` do `Kernel` que efetiva as alterações através do `WorldModel` invocando o método `merge`.

2.3 SWARM INTELLIGENCE

Existem na natureza espécies de insetos que vivem em colônias, denominados de insetos sociais (por exemplo, formigas, cupins, abelhas). Estes insetos, apesar de simples, são capazes de desenvolver tarefas complexas, por exemplo: a construção de formigueiros e colmeias; busca diária de alimentos para sobrevivência da colônia. Nestas colônias cada indivíduo tem uma classe e cada classe tem funções específicas dentro da colônia. Todas estas funções são necessárias para o objetivo da colônia que é a sobrevivência (BONABEAU, THERAULAZ, DORIGO, 1999, p. 1). Ainda segundo Bonabeau, Theraulaz e Dorigo (1999, p. 1), cada inseto social de uma colônia parece ter sua própria agenda (objetivo) e ainda, a integração das atividades individuais para formar a atividade da colônia como um todo parece não requisitar supervisão.

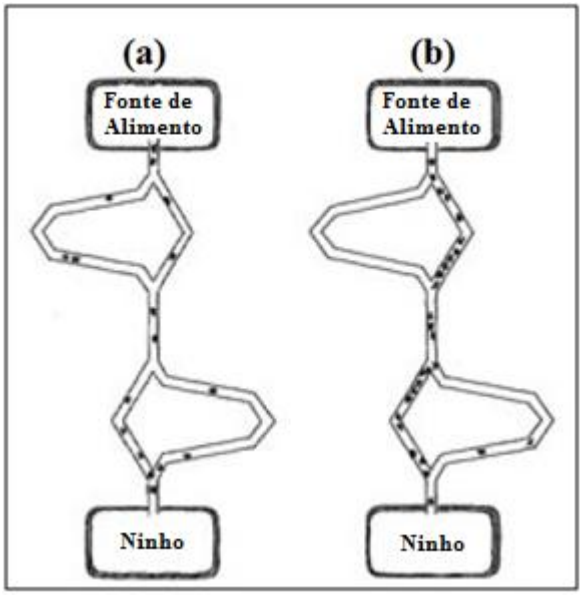
É deste princípio que parte a área de estudo da SI, também conhecida como Inteligência de Enxames (IE). A SI aplica os conceitos dos insetos sociais na área de IA e SMA. Estes conceitos são trazidos para a IA com o objetivo de facilitarem a resolução de problemas onde há ocorrência de muitos agentes e estes precisam trabalhar em conjunto e responder ao ambiente de alguma forma.

Um destes conceitos é o de comunicação pelo ambiente[1]. Este termo se refere à utilização do ambiente para comunicação entre indivíduos. Esta comunicação não ocorre por troca de mensagens diretamente, mas sim pelo ambiente, através de feromônios. Feromônio é uma substância química, que um indivíduo deposita no ambiente, e que é sentido por outros indivíduos. O feromônio depositado no ambiente evapora com o passar do tempo. O tipo de feromônio e também a quantidade existente transmitem a informação desejada, estabelecendo desta forma a comunicação (BONABEAU, THERAULAZ, DORIGO, 1999, p. 14).

A comunicação através do ambiente é verificada, por exemplo, nas formigas, quando se coloca uma fonte de comida separada do ninho das formigas por uma ponte de dois galhos aparentemente iguais. Inicialmente não há feromônio em nenhum dos dois galhos que têm a mesma probabilidade de serem selecionados pelas formigas. Eventualmente fatores aleatórios fazem com que algumas formigas a mais passem por um determinado galho, por exemplo, o galho “A” em vez do outro. Por haver mais formigas depositando feromônio enquanto andam no galho “A” as outras formigas sentirão mais estimuladas a passarem também pelo galho

[1] Em inglês o termo utilizado para definir este conceito é "stigmergy". Não existe termo equivalente em português.

"A". Ao se colocar galhos de comprimentos diferentes, como mostra a Figura 5 (a), as formigas escolhem inicialmente o caminho do mesmo modo que antes, aleatoriamente. Porém, após um tempo há uma diferença na escolha de um único caminho, Figura 5 (b). As formigas que escolheram o caminho mais curto são as que chegam antes ao outro lado e que voltam antes ao ninho, marcando o caminho percorrido com seus feromônios. Após estas formigas retornarem há mais concentração de feromônio no caminho mais curto do que no outro galho mais longo, pois ocorreu evaporação dos feromônios previamente depositados. Desta forma, as outras formigas são estimuladas a escolherem o galho mais curto também.



Adaptado de: Bonabeau, Theraulaz e Dorigo (1999, p. 29).

Figura 5 – Evolução do experimento com relação à escolha das formigas por meio de feromônios

Após diversas análises neste experimento Deneubourg et al. (1990, p. 159-168) desenvolveu um modelo matemático para representar este fenômeno, tendo por base que a quantidade de feromônio em um galho é proporcional ao número de formigas que passaram por ele (assumindo que cada formiga deposita uma unidade de feromônio). Assim, seus autores concluíram que a escolha de um caminho na ponte depende diretamente do número de formigas que passaram por este caminho.

Mais precisamente, tem-se F_{A_i}e F_{B_i} como a quantidade de feromônio depositada nos ramos *A* e *B* após *i* formigas usarem a ponte. A probabilidade P_A que a formiga (*i + 1*) escolha ramo *A* é modelada conforme a equação apresentada no Quadro 2.

$$P_A = \frac{\left(k + F_{A_i}\right)^n}{\left(k + F_{A_i}\right)^n + \left(k + F_{B_i}\right)^n}$$

Fonte: Bonabeau, Theraulaz e Dorigo (1999, p. 27).

Quadro 2 – Equação da probabilidade de escolha da formiga em relação aos feromônios nos caminhos

O parâmetro *n* determina o grau de não linearidade da função de escolha: quando *n* é grande, se um ramo tem apenas um pouco mais de feromônio do que a outro, a formiga que passa ali terá uma alta probabilidade de escolhê-lo. O parâmetro *k* quantifica o grau de atração de um ramo sem marcação: quanto maior *k*, maior será a quantidade de feromônio para fazer a escolha não aleatória.

A equação apresentada no Quadro 2 pode ser reescrita para denotar situações onde existem várias opções para o caminho escolhido pela formiga, como por exemplo, o esquema exposto na Figura 6.

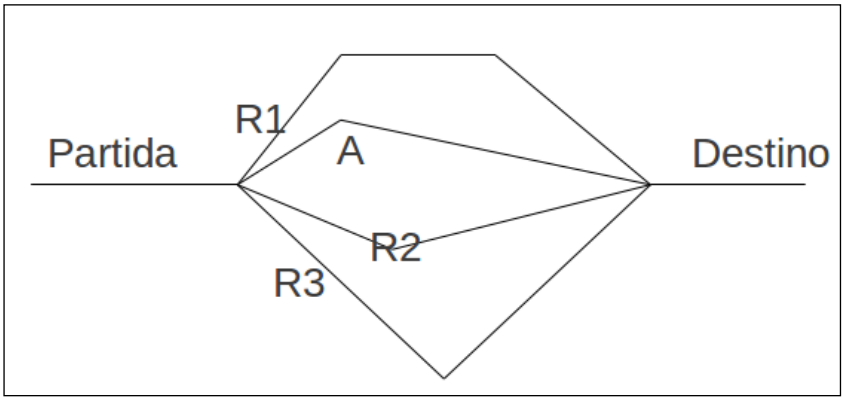


Figura 6 – Ilustração de situação com várias opções de caminho

Essencialmente, basta que o divisor da fração seja composto pelo somatório dos feromônios de todos os caminhos possíveis que seguem a partir de um determinado ponto, como mostra a equação no Quadro 3.

$$P_A = \frac{\left(k + F_{A_i}\right)^n}{\left(k + F_{A_i}\right)^n + \sum_R \left(k + F_{R_i}\right)^n}$$

Quadro 3 – Equação da probabilidade de escolha da formiga em relação ao caminho com feromônio, para vários caminhos

Em situações onde a quantidade de feromônio depositada por cada formiga no caminho é diferente de 1 unidade, Bonabeau, Theraulaz e Dorigo (1999, p. 43) afirmam que as quantidades depositadas por cada formiga devem ser somadas e adicionadas à quantidade existente no caminho, conforme apresenta a equação do Quadro 4.

$$\mathrm{F_R} = \mathrm{F_R} + \sum_{\mathrm{i}=0}^{\mathrm{m}} \Delta(\mathrm{F_i})$$

Fonte: Bonabeau, Theraulaz e Dorigo (1999, p. 43).

Quadro 4 – Equação para adição de feromônio em unidades maiores que 1

Nesta equação, *m* é a quantidade de formigas que passaram pelo caminho *R*, e $\Delta(F_i)$

é a quantidade de feromônio depositada pela formiga *i* no caminho.

Para modelar a evaporação de feromônio, Bonabeau, Theraulaz e Dorigo (1999, p. 43) propõem o uso de um coeficiente de evaporação, que seria responsável por determinar a quantidade de feromônio a ser evaporada do ambiente. A sua importância se deve ao fato de que através da evaporação do feromônio as formigas deixam de serem atraídas para um caminho que não é mais importante para a colônia. Por exemplo: uma rota para busca por alimento que deixou de ser usada, pois o alimento ali contido acabou.

A evaporação do feromônio F_R existente em um caminho *R* qualquer, é sugerida por Bonabeau, Theraulaz e Dorigo (1999, p. 43) conforme a equação no Quadro 5.

$$\mathrm{F_R} = (1 - \mathrm{p}) * \mathrm{F_R}$$

Fonte: Bonabeau, Theraulaz e Dorigo (1999, p. 43).
Quadro 5 – Equação para a evaporação de feromônio

Sendo *p* o coeficiente de evaporação, um número real que pode variar entre 0,1 e 1,0 e F_R a quantidade de feromônio presente no caminho. A equação define a nova quantidade de feromônio para o caminho após a evaporação.

2.4 TRABALHOS CORRELATOS

Um dos trabalhos correlatos encontrados foi desenvolvido por Kassabalidis et al. (2001) "*Swarm Intelligence for Routing in Communication Networks*". Este trabalho é uma pesquisa sobre um algoritmo de roteamento para redes baseado em conceitos de SI, mais precisamente na comunicação entre os agentes. A idéia do algoritmo é baseada na forma em como as formigas exploram o ambiente em busca de uma fonte de alimento e retornam ao ninho, deixando um rastro de feromônio. No algoritmo, chamado AntNet, é utilizado este princípio para construção de uma tabela de roteamento. Este tabela é montada a partir de agentes de exploração de rede que decidem qual seu próximo salto de forma aleatória até chegar a um nó específico, sendo que a decisão é influenciada pelos feromônios existentes na rota.

Outro trabalho correlato foi desenvolvido por Santos (2009) que apresenta um algoritmo que utiliza conceitos de SI para a resolução do problema de alocação de tarefas. O

conceito de alocação de tarefas vem do conceito de divisão de trabalho inspirado em estudo dos insetos sociais e no processo de recrutamento para transporte cooperativo presente em de algumas espécies de formigas.

> Através do modelo de divisão de trabalho, cada agente decide quais tarefas irá realizar, levando em consideração suas competências e recursos disponíveis. Ao perceber tarefas inter-relacionadas que requerem esforço simultâneo de um grupo de agentes, o agente reproduz o processo de recrutamento. [...] Agentes executando o *eXtreme-Ants* são eficientes para atuarem em *extreme teams*, pois requerem pouca comunicação e pouco esforço computacional. (SANTOS, 2009, p. 12).

O algoritmo foi desenvolvido para trabalhar com eficiência até mesmo em situações onde há restrições no ambiente como comunicação e tempo, ambiente este proporcionado no simulador RCR. Foi feita uma abordagem no simulador RCR utilizando este algoritmo para a divisão de tarefas entre os agentes de resgate. A eficiência desta abordagem, utilizando conceitos de SI, foi comprovada superior em relação à outras abordagens que não utilizam conceitos de SI considerando quesitos do próprio simulador, como *score* e tempo. Nas sugestões de extensões, Santos (2009) propõe que seja incorporado em seu algoritmo o uso de comunicação pelo ambiente para comunicação entre os agentes, possibilitando a eliminação do uso de comunicação direta.

3 DESENVOLVIMENTO DA EXTENSÃO

As seções seguintes descrevem a especificação e implementação da extensão SI do RCR. A operacionalidade será descrita através de uma implementação de SMA que utiliza a extensão. Por fim são indicados os resultados obtidos com este trabalho.

Para o desenvolvimento da extensão tomou-se por base a estrutura em que o simulador se encontrava atualmente, utilizando-se de recursos já implementados no simulador, como descrito nas seções 2.2.1 e 2.2.2.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A extensão SI deve incorporar no RCR o conceito de comunicação pelo ambiente, ou seja, comunicação entre os agentes através de feromônios. Para tanto a extensão deve oferecer um modo para que o agente possa depositar e ler os feromônios, de acordo com as equações descritas na seção 2.3. Se for possível para o agente depositar feromônios, será possível desenvolver um SMA que utilize estes conceitos para sua movimentação e assim realizar a comunicação pelo ambiente.

Além da possibilidade de depositar e ler os feromônios a extensão deve possuir um mecanismo para realizar a evaporação do feromônio, para que o caminho marcado perca a influencia do feromônio com o passar do tempo, conforme descrito na seção 2.3.

Os requisitos funcionais e requisitos não funcionais deste trabalho são apresentados nos quadros 6 e 7, respectivamente.

Requisitos Funcionais	**Casos de Uso**
RF01: manter as funcionalidades presentes no simulador (categorias de agentes, *score*, contagem de tempo).	UC01
RF02: permitir ao usuário desenvolver um SMA de simulação no RCR, que utilize funções de SI na programação dos agentes, em especial aquela que permite a comunicação através do ambiente.	UC02 e UC03

Quadro 6 – Requisitos funcionais

Requisitos Não Funcionais
RNF01: ser desenvolvido na linguagem de programação Java.
RNF02: funcionar em sistema Linux, padrão do simulador.

Quadro 7 – Requisitos não funcionais

3.2 ESPECIFICAÇÃO

Na sequência é apresentada a especificação da extensão, que foi modelada na ferramenta *Entreprise Architect* (SPARXSYSTEMS, 2000). Na especificação foram utilizados conceitos da *Unified Modeling Language* (UML) para a criação dos diagramas de caso de uso, de classes e de componentes.

3.2.1 Diagrama de casos de uso

Na especificação da extensão existem três cenários a serem atendidos. Os dois primeiros (Figura 7) têm como ator o usuário e o segundo (Figura 8) o *kernel*.

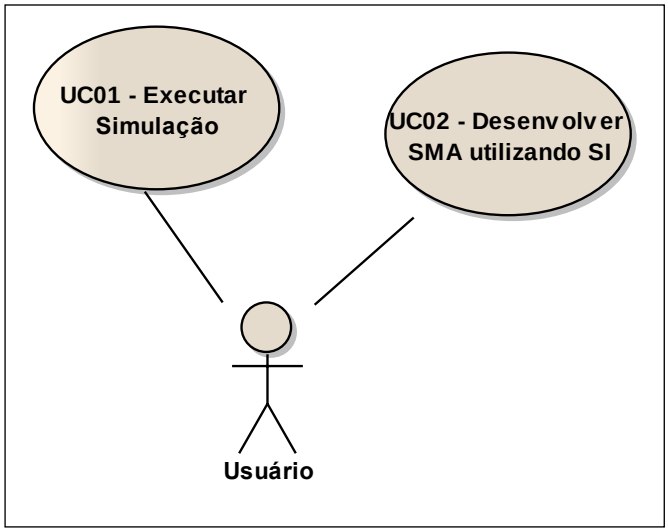


Figura 7 – Diagrama de casos de uso executado pelo usuário

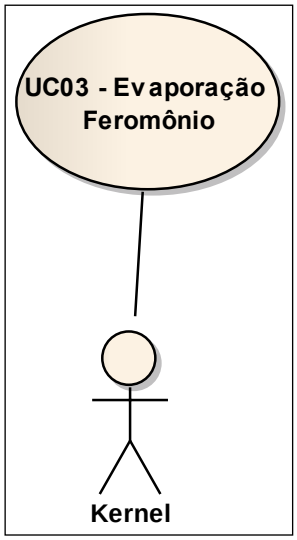


Figura 8 – Diagrama de casos de uso executado pelo *kernel*

O caso de uso `UC01 - Executar Simulação` (Quadro 8) define a execução da simulação pelo usuário. Para isto o usuário define parâmetros de configuração dos simuladores através de arquivos respectivos a cada simulador. Assim a leitura dos arquivos é efetuada e os valores de configuração atribuídos.

UC01 – Executar Simulação	
Requisito atendido	RF01.
Pré-condição	Componentes e agentes implementados.
Cenário Principal	1. O usuário define configurações e parâmetros (percentual de evaporação, *n* e *k*) para a execução da simulação. 2. O usuário inicia a simulação através dos scripts do simulador. 3. O RCR roda a simulação. 4. O usuário acompanha a execução da simulação. 5. O RCR termina a execução da simulação e apresenta o *score* ao usuário.
Pós-condição	Simulação realizada com sucesso.

Quadro 8 – Detalhamento do caso de uso `UC01 - Executar Simulação`

O caso de uso `UC02 - Desenvolver SMA utilizando SI` (Quadro 9) define como deve ser feita a implementação de um agente, pelo usuário, que utilize recursos de comunicação pelo ambiente disponibilizados pela extensão SI. No cenário são descritos os passos para esta implementação.

UC02 – Desenvolver SMA Utilizando SI	
Requisito atendido	RF02.
Pré-condição	Extensão SI disponível.
Cenário Principal	1. O usuário especifica a estratégia do agente, podendo utilizar comunicação pelo ambiente entre os agentes. 2. Ao implementar o agente, o usuário utiliza métodos disponíveis na extensão para: 2.1. Obter acesso à quantidade de feromônio depositada em cada rua; 2.2. Depositar feromônio nas ruas que compõem o trajeto que o agente se movimentará.
Pós-condição	SMA implementado.

Quadro 9 – Detalhamento do caso de uso `UC02 - Desenvolver SMA utilizando SI`

O caso de uso `UC03 - Evaporação de Feromônio` (Quadro 10) define a como será implementado o mecanismo de evaporação de feromônio, que será responsável por executar a dispersão do feromônio depositado pelo agente ao decorrer da simulação. A evaporação de feromônio é necessária para dispersar o feromônio presente nas ruas, que foi depositado pelos agentes. Assim quando uma rua não for marcada por certo tempo ela não terá mais a influencia do feromônio na escolha do caminho pelo agente.

UC03 – Evaporação de Feromônio	
Requisitos atendidos	RF02.
Pré-condições	Propriedade de feromônio disponível nas ruas. Parâmetro de evaporação de feromônio definido nas configurações
Cenário Principal	1. O mecanismo de evaporação lê o parâmetro de configuração para a evaporação do feromônio. 2. O mecanismo de evaporação itera sobre as ruas do mundo: 3. Em cada rua do mundo, o mecanismo de evaporação decrementa o valor do feromônio nela contido, de acordo com a equação descrita no Quadro 5 (fórmula da evaporação). 4. o mecanismo de evaporação adiciona as modificações ao `ChangeSet`, para que sejam visíveis pelo kernel.
Pós-condição	O feromônio presente na rua é decrementado.

Quadro 10 – Detalhamento do caso de uso `UC03 - Evaporação de Feromônio`

3.2.2 Especificação da Comunicação pelo ambiente

Para possibilitar o caso de uso `UC02 - Desenvolver SMA utilizando SI` são necessárias alterações na estrutura do simulador que permitam ao agente o depósito e a leitura do feromônio para a escolha do caminho.

Inicialmente havia sido definido que o depósito e a evaporação do feromônio seriam feitos em um único módulo, que receberia os comandos de depósito de feromônio dos agentes e executaria os comandos fazendo o depósito, além de calcular e efetuar a evaporação do feromônios das ruas. Na prática isto não foi possível devido ao fato de que o agente pode, por restrições do simulador RCR, enviar somente um comando de ação por *time-step*. Sendo assim, em um determinado *time-step*, ou o agente enviaria um comando de movimentação, ou enviaria um comando para depositar feromônio. Verificou-se que isto prejudicaria o desempenho dos agentes na simulação. Isto se deve ao fato de que o agente levaria o dobro do tempo para se locomover marcando o caminho, já que em um *time-step* seria enviado o comando referente à marcação do caminho e em outro o comando de movimentação. Enquanto um agente que não usaria a extensão levaria somente um *time-step* para se mover, enviando somente o comando de movimentação.

Em função disto, optou-se por escolher o simulador de tráfego para realizar o depósito de feromônio. Este simulador é responsável pela movimentação dos agentes no mapa, assim quando o agente envia um comando de movimentação, será indicado se é preciso ou não marcar as ruas depositando o feromônio.

A Figura 9 apresenta o diagrama com as classes do RCR que necessitam ser complementadas para possibilitar o processo da comunicação pelo ambiente.

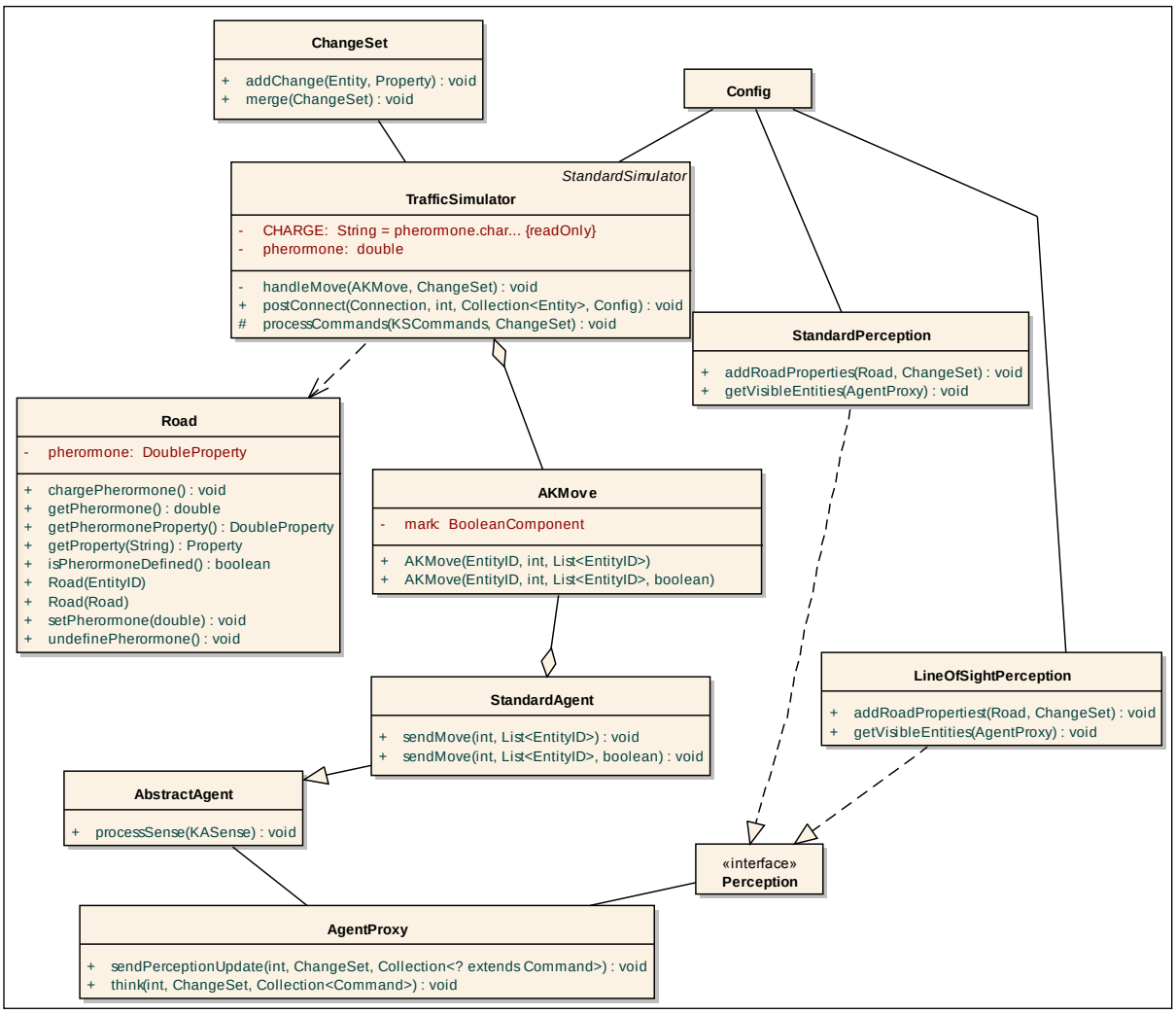


Figura 9 – Diagrama de classes com as classes alteradas no simulador

Inicialmente a classe `Road` necessita de um atributo que possa armazenar a quantidade de feromônio depositado nela (`pherormone`). Os agentes possuem acesso aos objetos de Road que estão dentro da sua percepção, logo, poderão acessar este atributo para ler a quantidade de feromônios existente. Para que o agente deposite feromônio ao se mover é necessária uma alteração na classe `AKMove`, responsável pelos comandos de movimentação, com um parâmetro booleano para indicar se as ruas do caminho enviado serão marcadas com feromônio. A tipificação deste parâmetro na mensagem já esta definida na estrutura do simulador como `BooleanComponent`, que representa um componente da mensagem do tipo booleano. Para possibilitar o envio desta mensagem a classe `StandardAgent` necessita da inclusão de um método que aceite o novo parâmetro.

A classe `TrafficSimulator` deve ser alterada para poder tratar o novo comando e marcar com feromônio as ruas por onde o agente escolheu passar. O método da classe

responsável pelo tratamento deste comando é o `handleMove`. Nele serão feitas as alterações para o tratamento do comando e o depósito de feromônio, de acordo com a equação presente no Quadro 4.

A quantidade de feromônio a ser depositada na rua pelo método da classe `TrafficSimulator` será definida através de um parâmetro de configuração onde seu valor será lido a partir do arquivo de configuração *traffic.cfg*. O valor em especial é definido no arquivo por um parâmetro denominado `pherormone.charge_coef`. Este parâmetro é um valor fixo associado ao atributo `pherormone` sendo responsável pela quantidade de feromônio que cada agente deposita na rua.

Como pode ser visto no diagrama de sequência (Figura 10) o simulador de tráfego recebe os comandos enviados do *kernel*, através do `SimulatorProxy`, que os repassa pelo método `processCommands`. Ao receber os comandos o simulador de tráfego identifica se algum comando é pertinente a ele, se for é feito o processamento do mesmo. No caso, o comando de deslocamento `AKMove` é filtrado e tratado pelo método `handleMove`. Ao tratar o comando e identificar a necessidade de marcar as ruas, o método adiciona o feromônio na rua de acordo com o parâmetro configurado. Depois de adicionar a quantidade de feromônio é gravado no `ChangeSet` a modificação feita na rua. Após o simulador ter terminado a execução dos comandos e alterado os objetos mundo ele retorna para o `kernel` o `ChangeSet` preenchido com todas as suas modificações. Então o `kernel` executa o método `merge` de seu `ChangeSet` para agrupar todas as modificações dos simuladores em um único `ChangeSet`. Com isto feito o `kernel` efetiva as modificações invocando o método `merge` do `WorldModel`, passando este `ChangeSet` unificado como parâmetro, para que a quantidade de feromônio seja atualizada na representação do mundo mantida pelo *kernel*.

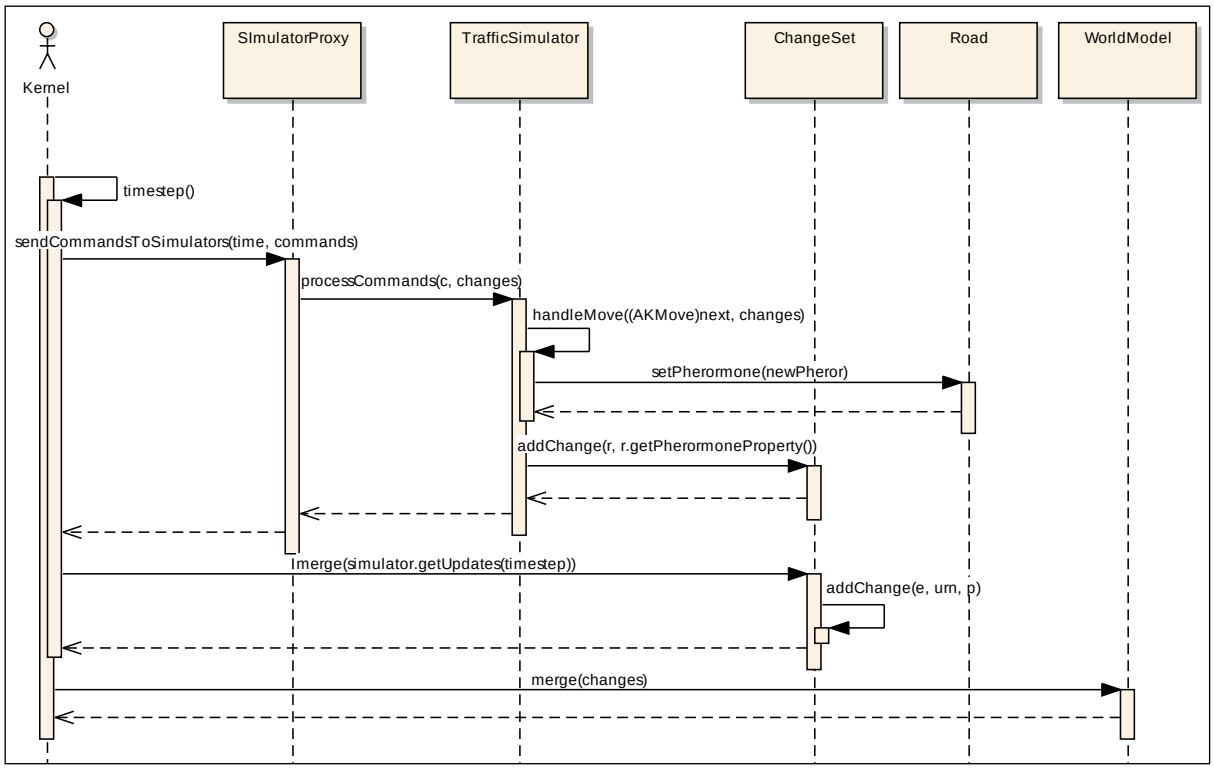


Figura 10 – Diagrama de sequência representando o processo de depósito de feromônio

Para que o agente possa perceber o valor do atributo `pheromone` presente na rua é preciso que o mesmo seja adicionado à percepção gerenciada pelo `Kernel`. Assim, as classes `StandardPerception` e `LineOfSightPerceptio`, que representam a percepção dos agentes, devem ser alteradas para que o novo atributo possa ser reconhecido pelo agente e alterado pelo mesmo. A Figura 11 apresenta o processo de atualização da percepção, feito pelo `Kernel` e intermediado pelo `AgenteProxy`.

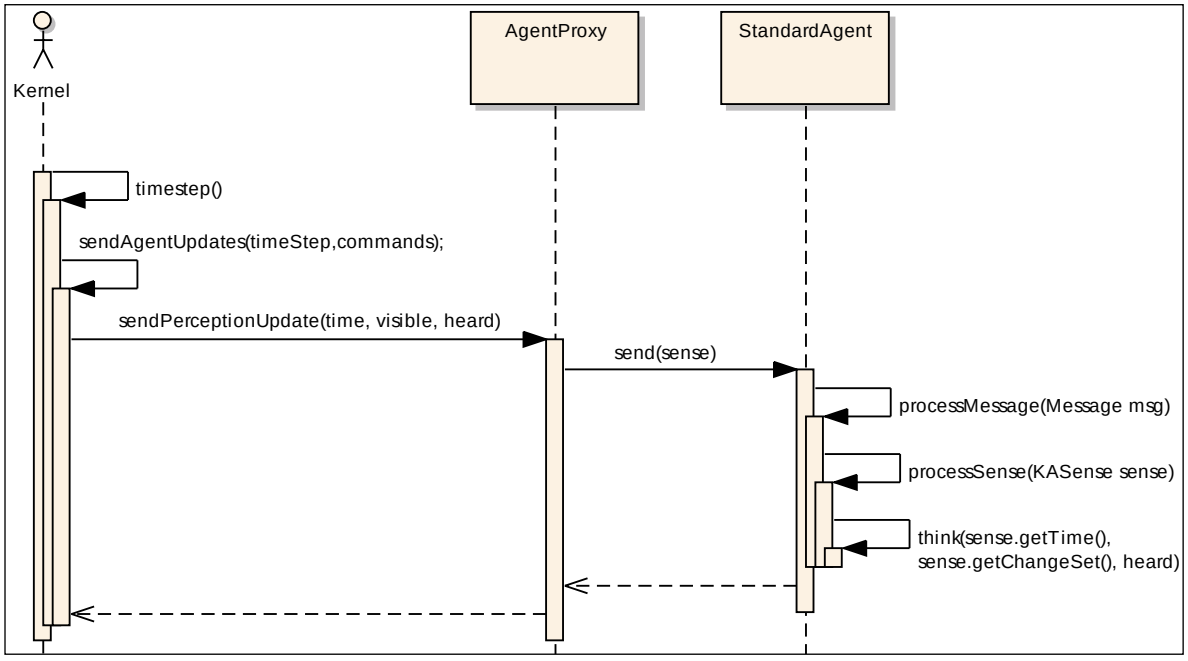


Figura 11 – Diagrama de sequência representando a atualização da percepção do agente

Durante o *time-step* o `kernel` atualiza a percepção dos agentes implementados através do método `sendAgentsUpdates`, neste método o `kernel` itera sobre os agentes através do `AgentProxy` e invoca o método `sendPerceptionUpdate`, do próprio `AgentProxy`. Este método, `sendPerceptionUpdate`, é responsável por enviar uma mensagem, `send(sense)`, representando a nova percepção do agente, `sense`. Quando o agente recebe esta mensagem, através do método `processMessage`, ele a interpreta, atualizando sua percepção no método `processSense`. De acordo com a nova percepção processa suas ações através do método `think`.

3.2.3 Especificação da Evaporação de Feromônio

Para atender ao caso de uso `UC03 - Evaporação de feromônio` é necessário a especificação de um simulador que fique responsável por esta tarefa. Este simulador será chamado de *PheromoneSimulator* e será um componente associado ao `Kernel` da mesma forma que os outros simuladores, como apresentado no diagrama de componentes da Figura 12.

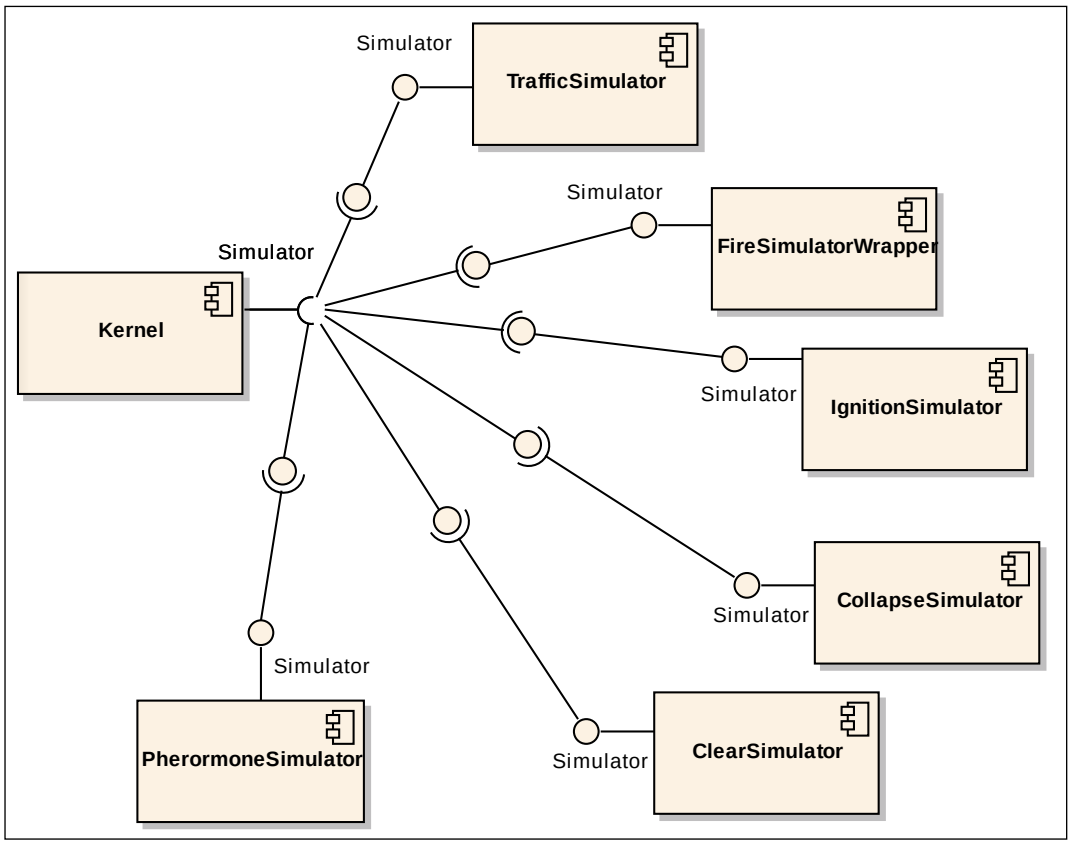


Figura 12 – Diagrama de componentes com o `PherormoneSimulator`

A Figura 13 apresenta o digrama de classes do componente *PherormoneSimulator*. A classe `PheromoneSimulator` estende o `AbstractSimulator` que se relaciona com o `WorldModel` dando acesso ao mundo para alterar a propriedade de feromônio presente nas ruas. O processo de evaporação de feromônio executado pelo simulador depende exclusivamente da existência das ruas, representadas pela classe `Road`, com o atributo que representa o feromônio.

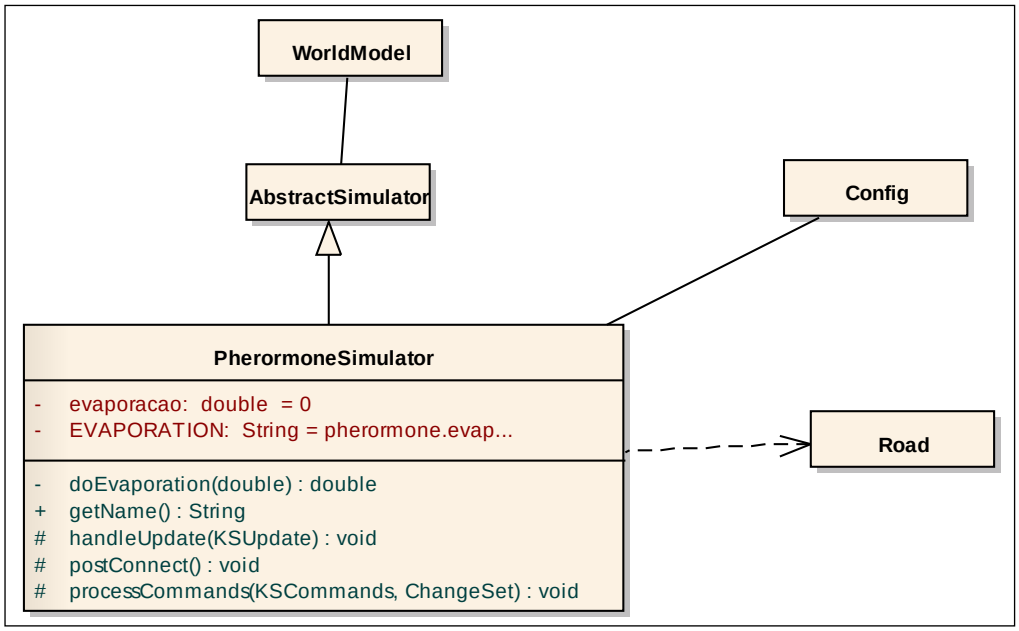


Figura 13 – Especificação do componente `PherormoneSimulator`

O simulador ficará responsável por gerenciar a evaporação do feromônio contido nas `Roads` a cada *time-step*. Para executar isto o simulador deverá varrer todas as ruas e diminuir uma porcentagem do feromônio contido na rua. Esta porcentagem é pré-definida através de um arquivo de configuração, denominado `pherormone.cfg`, e que deve ser informado como parâmetro para o `PherormoneSimulator`. O valor a ser estipulado para a evaporação é indicado pelo parâmetro `pherormone.evaporation_coef` dentro do arquivo de configuração. A classe `Config` é a responsável pela leitura do arquivo de configuração do simulador passando assim os valores aos atributos da classe.

A figura 14 apresenta o diagrama de sequência da evaporação de feromônios. O simulador é invocado pelo método `processCommands` e então itera sobre as ruas do mapa decrementando a quantidade de feromônio presente em cada uma delas. O cálculo da quantidade a ser evaporada é feito pelo método `doEvaporation` e retornada ao método `processCommands`. Assim, o novo valor do atributo é definido para a rua, representada pela classe `Road`. Após o simulador ter terminado a execução dos comandos e alterado os objetos do mundo ele retorna para o `kernel` o `ChangeSet`. Então o `kernel` agrupar todas as

modificações dos simuladores em um único `ChangeSet` e efetiva as modificações invocando o método `merge` do `WorldModel`.

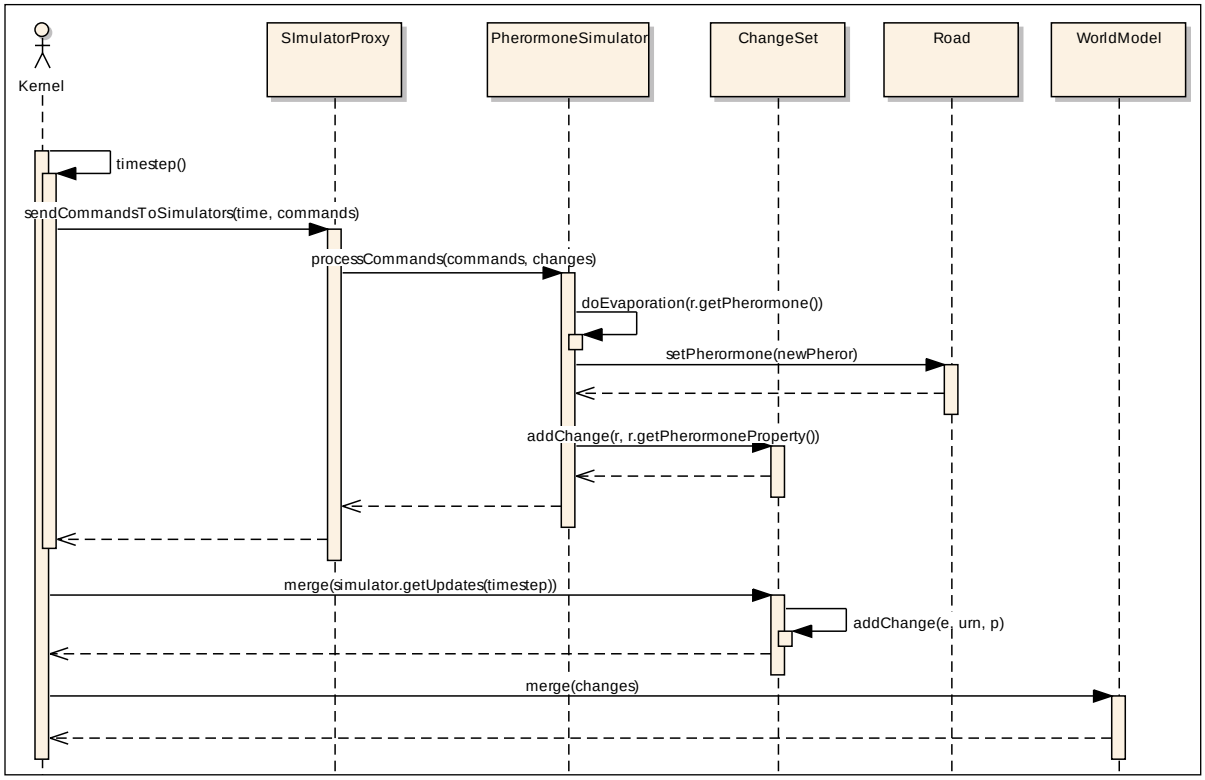


Figura 14 – Diagrama de sequência representando a evaporação de feromônios

Devido ao fato de dois simuladores poderem alterar a mesma propriedade de um objeto, ao mesmo tempo, precisou-se adicionar um tratamento diferenciado ao se adicionar a modificação para a propriedade de feromônio presente nas ruas. Este tratamento foi adicionado através de uma comparação que verifica se a propriedade alterada era o feromônio, no método `addChange` da classe `ChangeSet`.

O método atual compreendia que a modificação de um atributo poderia ser feita apenas por um simulador, se outro simulador modificasse o mesmo atributo a mudança seria sobre-escrita pela nova. Então de acordo com o tratamento seria calculada a diferença resultante das modificações dos simuladores para ser definido o novo valor da propriedade.

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação da extensão, bem como o processo de implementação.

3.3.1 Técnicas e ferramentas utilizadas

A extensão SI foi implementada na linguagem de programação Java, na qual já se encontrava implementado o simulador, seguindo o paradigma de orientação a objetos, também adotado pelo simulador. Utilizou-se o ambiente de desenvolvimento Eclipse (ECLIPSE, 2008) por se ter maior domínio do mesmo.

3.3.2 Desenvolvimento da Comunicação pelo ambiente

Para o desenvolvimento da comunicação pelo ambiente foram alteradas as classe do simulador conforme descrito na seção 3.2.2.

A classe `TrafficSimulator` (representa o simulador de tráfego) foi alterada para poder interpretar, através do comando `AKMove`, quando fosse necessário marcar a rua com feromônio. Dentro do simulador os comandos são recebidos e filtrados no método `processComands`. De acordo com o comando recebido, se ele for para o simulador em específico, um método é chamado para interpretá-lo. No caso do `AKMove` o método chamado é o `handleMove`.

Primeiramente foi adicionado, junto ao parâmetro `AKMove` que contém o caminho escolhido pelo agente, o parâmetro `ChangeSet` para que fosse possível adicionar modificações através deste método. O Quadro 11 exibe a alteração.

```
1.   private void handleMove(AKMove move, ChangeSet changes){
2.   //omitido código do simulador que é responsável por iterar sobre o
3.   //caminho recebido e definir a movimentação do agente por ele.
4.      if(move.getMark()){
5.          //o método getEntity retorna o objeto identificado pela id
6.          // que representa do nodo do caminho atual (current)
7.          Entity ent = model.getEntity(current);
8.          if(ent instanceof Road){
9.             Road r = (Road) ent;
10.            double pheror = r.getPherormone();
11.            double newPheror = pheror + pherormone;
12.            r.setPherormone(newPheror);
13.            //Adiciona modificação ao changeSet.
14.            changes.addChange(r, r.getPherormoneProperty());
15.         }
16.     }
17.  }
```

Quadro 11 – Trecho de código adicionado ao método `handleMove`

O método itera sobre a lista de caminhos passada no comando `AKMove` e para cada nodo da lista verifica a possibilidade de marcação da rua do caminho com feromônio, através do parâmetro `mark`. Após verificar se é preciso marcar a rua com feromônio, é validado se o

nodo do caminho representa uma rua, pois é somente na rua que a marcação de feromônio ocorre. Na linha 10 é lida a quantidade de feromônio atual da rua e na linha 11 é calculada a nova quantidade de feromônio da rua de acordo com a quantidade depositada pelo agente. Esta quantidade é fixa e representa o valor de $\Delta(F_i)$ da equação do Quadro 4, que define o cálculo da adição de feromônio. A quantidade de feromônio a ser depositada é estipulada através de um arquivo de configuração que é associado ao atributo `pherormone` pela classe `Config`. O nome do arquivo é informado ao inicializar o simulador de feromônio e este arquivo de configuração será lido com a criação de uma instancia da classe `config` que contenha os parâmetros definidos no arquivo.

Para que fosse possível ao agente enviar este comando, foi necessário incluir um método (Quadro 12) na classe `StandardAgent`, que construísse uma mensagem com o atributo *mark*. Este método pode então ser chamado pelo usuário, na implementação de seu agente.

```
protected void sendMove(int time, List<EntityID> path, boolean mark) {
   send(new AKMove(getID(), time, path, mark));
}
```

Quadro 12 – Novo método para o envio da mensagem `AKMove`

As alterações para possibilitar a percepção da propriedade do feromônio presente nas ruas pelo agente se limitaram a inclusão do código, apresentado no Quadro 13, nas classes `StandardPerception` e `LineOfSightPerception`. Dentro do método `addRoadProperties` é necessário adicionar a propriedade do feromônio no `ChangeSet` da percepção do agente. Desta forma, a percepção do agente é atualizada com o novo valor do feromônio da rua.

```
private void addRoadProperties(Road road, ChangeSet result) {
//omitido código do simulador que é responsável por adicionar demais
//propriedades da rua
        //atualiza feromônios
        result.addChange(road, road.getPherormoneProperty());
}
```

Quadro 13 – Código que adiciona a propriedade de feromônio da rua na percepção do agente

3.3.2.1 Desenvolvimento da Evaporação de Feromônio

Para a implementação do simulador de evaporação foi tomado como base a implementação de simuladores já existentes. A classe implementada foi chamada de `PherormoneSimulator`. Sua função é apenas varrer as ruas do mundo e decrementar uma porcentagem de feromônio de acordo com o valor estabelecido por parâmetro no arquivo de configuração `pherormone.cfg`. O conteúdo do arquivo de configuração deve ser definido

conforme mostra o Quadro 14.

```
1. !include common.cfg
2. pherormone.evaporation_coef:25
```

Quadro 14 – Arquivo de configuração do simulador de evaporação de feromônio

Na linha 1 são incluídas configurações comuns a todos os simuladores, no quesito de conexão com o `kernel`, como porta e o host para a conexão. Na segunda linha é definido o valor referente ao parâmetro que corresponde ao coeficiente de evaporação (`pherormone.evaporation_coef`) com o valor de 25. Este valor representa a porcentagem de evaporação que será calculada no simulador, neste caso 25%.

A definição e inicialização da classe `PherormoneSimulator` é apresentada no Quadro 15.

```
public class PherormoneSimulator extends StandardSimulator{
    //Define parâmetro a ser lido no arvido de configuração
    private static final String EVAPORATION = "pherormone.evaporation_coef";
    //Parametro com o valor do percentual de evaporação
    private double evaporacao = 0;
    public String getName() {        return "Basic Pherormone Simulator";    }
    protected void postConnect() {
        //Executa a conexão com o kernel
        super.postConnect();
        //Atribui o valor do parâmetro do arquivo de configuração
        evaporacao = config.getIntValue(EVAPORATION);
    }
    //método chamado pelo kernel.
    protected void processCommands(KSCommands c, ChangeSet changes){    }
    //Efetua o cálculo da diminuição do feromônio da rua.
    private double doEvaporation(double pherormone){
        return pheror = pherormone - (pherormone*(evaporacao/100));
    }
}
```

Quadro 15 – Classe `PherormoneSimulator`

Conforme apresenta o Quadro 16, no método `processCommands` o simulador itera sobre todas as ruas, executando a evaporação do feromônio de cada uma delas. Este cálculo da quantidade de feromônio a ser decrementada, representando a evaporação, é feito pelo método `doEvaporation` (linha 9), definido no Quadro 15. Após isto as modificações são adicionadas ao `ChangeSet` representado pelo atributo `changes`.

```
1.  //método chamado pelo kernel.
2.  protected void processCommands(KSCommands c, ChangeSet changes){
3.     //Lê as ruas do mundo
4.     Collection<StandardEntity> e =
       model.getEntitiesOfType(StandardEntityURN.ROAD);
5.     for (StandardEntity next : e) {
6.        if (next instanceof Road) {
7.           Road r = (Road) next;
8.           //para cada rua evapora quantidade de feormonio.
9.           double newPheror = doEvaporation(r.getPherormone());
10.          r.setPherormone(newPheror);
11.          //Adiciona modificação ao changeSet.
12.          changes.addChange(r, r.getPherormoneProperty());
13.       }
14.    }
15. }
```

Quadro 16 – Método `processCommands` da classe `PherormoneSimulator`

O tratamento adicionado ao método `addChange` da classe `ChangeSet`, descrito na seção de especificação, responsável pela junção das modificações dos simuladores é exibido no Quadro 17.

```
public void addChange(EntityID e, String urn, Property p) {
    if (deleted.contains(e)) {
       return;
    }
    Property prop = p.copy();
    if( prop.getURN().toString().equals(
                                    "urn:rescuecore2.standard:property:pherormone"
                                  )){
       DoubleProperty novoProp = (DoubleProperty) prop;
       double novo = novoProp.getValue();
       DoubleProperty antigoProp =
                               (DoubleProperty) getChangedProperty(e, p.getURN());

       if(antigoProp != null){
          double antigo = antigoProp.getValue();
          double diferenca = antigo - novo;
          novoProp.setValue(novo+diferenca);
          prop = novoProp;
       }
    }
    changes.get(e).put(prop.getURN(), prop);
    entityURNs.put(e, urn);
}
```

Quadro 17 – Função modificada responsável por adicionar as modificações

No caso, para saber se a propriedade é o feromônio compara-se a `String` com identificador da propriedade e então é feito o cálculo sobre o novo e o antigo valor, já definido, para determinar a diferença que será o novo valor a ser atribuído à propriedade.

3.3.3 Operacionalidade da implementação

Esta seção tem como objetivo mostrar a operacionalidade da extensão, abordando o atendimento ao caso de uso `UC02 - Desenvolver SMA utilizando SI`. Para demonstração da operacionalidade foi adotado para testes somente o agente bombeiro, por ter mais semelhança em sua tarefa com as formigas. No caso uma formiga sai do ninho em busca do alimento, recolhe o alimento e retorna ao ninho para depois sair novamente em busca de alimento. O agente bombeiro varre o mapa em busca de um incêndio, usa a água, contida em seu tanque para apagar o incêndio e quando a água acaba ele vai para o refúgio se abastecer com mais água para retornar e apagar o incêndio.

3.3.3.1 Especificação do Agente

A especificação das classes que compõem a implementação do agente é exibida na Figura 15. A classe `AbstractPherorAgent`, que foi estendida da classe `StandardAgent`, é responsável por definir a conexão dos agentes com o `kernel`, ler o arquivo de configuração com os parâmetros dos agentes, chamado `pheror-agent.cfg`, instanciar o algoritmo de busca de caminhos dos agentes e registrar as posições dos refúgios. Nela também esta implementado o método de busca por um caminho aleatório (`ramdomWalk`). O algoritmo de busca de caminhos é definido pela classe `PherormoneSearch`.

O arquivo de configuração irá conter como parâmetros as constantes *k* (`search.const_k`) e *n* (`search.const_n`), utilizadas na fórmula desenvolvida por Deneubourg et al. (1990, p. 159-168) apresentada nos Quadros 1 e 2. Este arquivo também irá conter o valor da distância limite (`search.const_limite`) para a busca por feromônios. Os valores lidos serão associados a atributos da classe `PherormoneSearch`, responsável pelos algoritmos de busca de caminho do agente. Quando o valor de *k* for 0 , o agente adota o valor `Float.MIN_Value`, que não tem influência relevante no valor obtido e visa evitar divisões por zero. A distância limite tem a função de limitar a distância que o agente irá percorrer por busca.

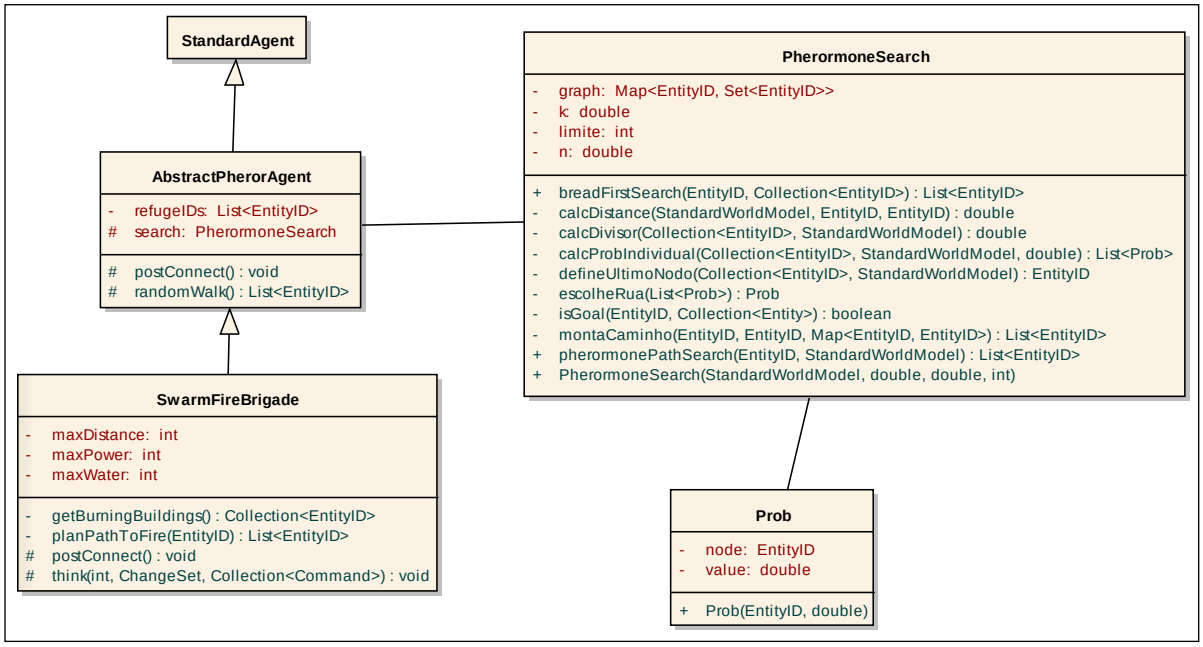


Figura 15 – Especificação da implementação do agente

A classe `Prob` foi criada para representar uma estrutura simples que armazenasse o nó do caminho e a sua probabilidade, conforme requer a fórmula apresentada no Quadro 3.

A estratégia utilizada para a implementação do agente foi baseada no sentido de que o agente bombeiro, que será chamado de *SwarmFireBrigade*, necessita primeiramente de água para apagar algum incêndio. Tendo que todos os agentes iniciam com o tanque cheio de água, se o agente estiver sem água em seu taque, é porque ele estava apagando um incêndio. Então o *SwarmFireBrigade* se move para o refúgio mais próximo dele, sem buscar o caminho através de feromônios. Isto não se faz necessário porque o agente tem registrado a localização de todos os refúgios no mapa, não precisando assim, procurar pelo local. Somente é feita a marcação do caminho que percorre até o refúgio com feromônio. Com isto, os agentes que estiverem saindo do refúgio, não tendo nenhum incêndio por perto para apagar, poderão encontrar o incêndio cujo caminho foi marcado. Ao terminar de encher o tanque, o agente verifica se não há nenhum incêndio em seu campo de visão, que é definido pela sua percepção explicada na seção 2.2.1. Se perceber um incêndio próximo o *SwarmFireBrigade* vai em direção a ele para apagá-lo. Caso contrário, utiliza a estratégia de movimentação por comunicação pelo ambiente, verificando os feromônios na rua para calcular o caminho a ser seguido de acordo com a equação de probabilidade mostrada no Quadro 3.

O *SwarmFireBrigade* define sua movimentação a partir da estratégia de comunicação pelo ambiente, verificando os feromônios na rua para calcular o caminho a ser seguido de acordo com a equação de probabilidade mostrada no Quadro 3.

O Quadro 18 apresenta pseudocódigo do que o agente deve fazer.

```
se não está cheio de água e esta no refúgio então
      enche água
      retorna
fimse

se está sem água então
      move para refúgio mais próximo marcando o caminho
      retorna
fimse

procura incêndio no campo de visão
se está perto o bastante então
      apaga incêndio
      retorna
senao
      move para apagar
      retorna
fimse

se não achou nada então
      move usando comunicação pelo ambiente.
      retorna
fimse
```

Quadro 18 – Pseudocódigo das ações do agente

3.3.3.2 Implementação do Agente

A classe que implementa o agente definido na especificação é chamada de `SwarmFireBrigade`. Sua declaração de atributos, implementação do método como nome do agente e a implementação da conexão do agente são exibidos no Quadro 18.

```
public class SwarmFireBrigade extends AbstractPherorAgent <FireBrigade> {
    private static final String MAX_WATER_KEY="fire.tank.maximum";
    private static final String MAX_DISTANCE_KEY="fire.extinguish.max-distance";
    private static final String MAX_POWER_KEY ="fire.extinguish.max-sum";

    private int maxWater;
    private int maxDistance;
    private int maxPower;

    @Override
    public String toString() {
        return "Agente SwarmFireBrigade ";
    }

    @Override
    protected void postConnect() {
        super.postConnect();
        model.indexClass(StandardEntityURN.BUILDING, StandardEntityURN.REFUGE);
        maxWater = config.getIntValue(MAX_WATER_KEY);
        maxDistance = config.getIntValue(MAX_DISTANCE_KEY);
        maxPower = config.getIntValue(MAX_POWER_KEY);
        //Exibe no log a confirmação de conexão.
        Logger.info("Swarm Fire Brigade connected: max extinguish distance = "+
                    maxDistance + ", max power = " + maxPower +
                    ", max tank = " + maxWater
                   );
    }
```

Quadro 19 – Trecho de código com parte da definição da classe `SwarmFireBrigade`

No método `postConnect` são atribuídos os valores de configuração para o bombeiro no que diz respeito à função de apagar incêndios. Estes valores em questão representam a capacidade máxima de água do tanque do agente (`maxWater`), a distância máxima que o agente pode ficar para apagar um incêndio (`maxDistance`) e a quantidade de água que ele joga em um incêndio por ciclo (`maxPower`).

O Quadro 20 apresenta outra parte da implementação da classe do agente `SwarmFireBrigade`. O método apresentado é o `think`, onde estão definidas as ações dos agentes. Este método corresponde à implementação Java do pseudocódigo apresentado no Quadro 17.

```
protected void think(int time, ChangeSet changed, Collection<Command> heard){
   FireBrigade me = me();
   // Estou me enchendo de água?
   if (me.isWaterDefined() && me.getWater() <
        maxWater && location() instanceof Refuge
       ){
         Logger.info("Enchendo com Água em: " + location());
         sendRest(time);
         return;
   }
   // Estou sem Água?
   if (me.isWaterDefined() && me.getWater() == 0) {
       // Procurar por um refúgio
       List<EntityID> path = search.breadthFirstSearch( me().getPosition(),
                                                        refugeIDs
                                                      );
            if (path != null) {
                //move marcando o caminho com feromônio
                sendMove(time, path, true);
                return;
            }
            else {
                //Se não encontrou caminho se move aleatóriamente
                path = randomWalk();
                sendMove(time, path);
                return;
            }
        }
        // Procura construções com incêndio
        Collection<EntityID> all = getBurningBuildings();
        // Alguma Próxima o Bastante?
        for (EntityID next : all) {
            if (model.getDistance(getID(), next) <= maxDistance) {
                Logger.info("Apagando Incêndio em: " + next);
                sendExtinguish(time, next, maxPower);
                return;
            }
        }
        //Se não há próxima o bastante, define caminho para o incêndio que viu
        for (EntityID next : all) {
            List<EntityID> path = planPathToFire(next);
            Logger.info("Movendo para incêndio");
            sendMove(time, path);
            return;
        }
        //Se não encontrou nada
        Logger.debug("Sem incêndios por perto");
         Logger.info("Procurar caminho por Feromônio.");
        List<EntityID> pathpheror = search.pherormoneSearch(
                                                          me().getPosition(), model
                                                        );
        if (pathpheror.size() > 1) {
             Logger.info("Movendo com Feromônio - > "+ pathpheror );
             sendMove(time, pathpheror);
             return;
        }
    }
```

Quadro 20 – Trecho de código com o método `think` do `SwarmFireBrigade`

Neste método é implementada a estratégia definida anteriormente na especificação do agente (Seção 3.3.3.1). Primeiramente o agente verifica se precisa encher seu tanque de água e está em um refúgio. Se sim ele envia o comando `sendRest` que sinaliza que o agente está em um refúgio e que ele pode ser abastecido com água. Após enviar este comando é utilizado o comando `return` para sair da função e terminar a ação do agente neste *time-step*.

Caso o agente não esteja em um refúgio e precise se abastecer de água, o agente procura pelo refúgio mais próximo do ponto onde ele se encontra, sem utilizar comunicação pelo ambiente. Para isso é utilizada o método `breadthFirstSearch` que irá retornar o caminho para o refúgio a ser seguido pelo agente. Com o caminho definido o agente envia o comando de movimentação `sendMove` com o parâmetro que indica a marcação do caminho e encerra sua ação. Se o agente não conseguir definir um caminho para o refúgio ele define um caminho aleatório através da função `randomWalk` determinada pela classe `AbstractPherorAgent` e encerra sua ação.

Sendo as condições anteriores não atendidas, o agente procura por um incêndio em seu campo de visão, através do método `getBurningBuildings()`. Se a distância entre o agente e o incêndio estiver dentro da capacidade do agente jogar água ele apaga o incêndio, Caso contrário, o agente se desloca para mais próximo do incêndio.

Por fim, caso não haja nenhum incêndio no campo de visão do agente, ele se move utilizando a estratégia de movimentação através da comunicação pelo ambiente. Esta estratégia é definida no método de busca `pherormoneSearch`, que será detalhada a seguir. São passados como parâmetros para esta função a posição inicial do agente e o modelo do mundo, representando o mapa com os objetos (`WorldModel`).

Após definir o caminho por feromônios através da função de busca, é enviado o comando `sendMove` sem o parâmetro de marcação do caminho, para que o agente desloque-se pelo caminho escolhido

O Quadro 21 exibe os métodos privados `getBurningBuildings` e `planPathToFire` utilizados pelo agente para encontrar incêndios e deslocar-se até ele sem comunicação pelo ambiente.

```
private Collection<EntityID> getBurningBuildings() {
Collection<StandardEntity> e=model.getEntitiesOfType(StandardEntityURN.BUILDING);
        List<Building> result = new ArrayList<Building>();
        for (StandardEntity next : e) {
            if (next instanceof Building) {
                Building b = (Building)next;
                if (b.isOnFire()) {  result.add(b); }
            }
        }
        //Ordena de acordo com a distância
        Collections.sort(result, new DistanceSorter(location(), model));
        return objectsToIDs(result);
}
private List<EntityID> planPathToFire(EntityID target) {
   // Tenta encontrar algum incêndio dentro do campo de visão
   Collection<StandardEntity> targets = model.getObjectsInRange(
                                                        target, maxDistance
                                                           );
   if (targets.isEmpty()) {  return null;  }
   return search.breadthFirstSearch(me().getPosition(), objectsToIDs(targets));
}//fim método
}//fim classe
```

Quadro 21 – Métodos do agente para encontrar incêndios

O método `getBurningBuildings` itera sobre as construções, verifica quais tem incêndios e grava em uma lista ordenada pela distância em relação à posição do agente. Já o método `planPathToFire` lê do mundo os objetos em volta do destino determinado pelo agente em um campo limitado pela sua capacidade máxima de lançar água. Após isto é invocado o método de busca direta de caminhos `breadthFirstSearch` a partir da posição atual até algum destes objetos lidos anteriormente. O método `breadthFirstSearch` que tem como função definir o caminho mais curto entre um ponto e outro. Para isso o método lê o mapa dos vizinhos de cada nó, a partir do nó inicial, até chegar ao destino. Este método não utiliza o conceito de comunicação pelo ambiente.

3.3.3.2.1 Algoritmo de busca de caminho por feromônio

Para efetuar a definição do caminho a ser seguido pelo agente foi implementado um algoritmo de busca aplicando a fórmula apresentada no Quadro 2. A fórmula representa uma abordagem do cálculo da probabilidade de escolha do caminho para um cenário onde o agente tem várias opções de escolha.

O Quadro 21 apresenta a declaração do método `pherormonePathSearch`, responsável pela busca de caminho influenciado por feromônio. O método recebe como entrada o nó inicial do caminho, a posição atual do agente, juntamente com o modelo do mundo com a situação dos objetos naquele *time-step*.

```
public List<EntityID> pherormonePathSearch(
                                        EntityID start,StandardWorldModel model
                                       ){
   List<EntityID> open = new LinkedList<EntityID>();
   Map<EntityID, EntityID> ancestors = new HashMap<EntityID, EntityID>();
   open.add(start);
   EntityID next = null;
   boolean found = false;
   ancestors.put(start, start);
   //Lista de nós já utilizados
   prev = new LinkedList<EntityID>();
   //divisor da fórmula .
   double divisor;
   //contador da distância percorrida
   double distância = 0;
   //estrutura para armazenar as probabilidades para cada rua.
   List<Prob> probs;
   do{    //iteração para a montar o caminho
   }while (!found && !open.isEmpty());
   // Não encontrou caminho
   if (!found) { return null; }
   List<EntityID> path =  montaCaminho(start, next, ancestors);
   return path;
}//fim função
```

Quadro 22 – Método da busca de caminho por feromônio

Inicialmente são definidas as estruturas para que o caminho seja "montado". A lista

`open` será responsável por armazenar o nó que será iterado na busca por um caminho, a variável `next` representa o próximo nó a ser iterado, o *hash* `ancestors` será responsável por armazenar a sequência entre os nós do caminho, que são identificadores das ruas do mapa. A lista `prev` guardará os nós já escolhidos para que o caminho não seja repetido. O método `montaCaminho` é responsável por ler o *hash* `ancestors` e montar o caminho partindo do inicio até o fim. Para armazenar os nós escolhidos anteriormente foi definida a lista `prev` e para os nós rejeitados, que não são ruas, foi definida a variável `reject`. Estas duas variáveis são globais para a classe já que são utilizadas nos demais métodos envolvidos no processo da busca por feromônio.

A iteração, responsável por montar o caminho, é apresentada no Quadro 23.

```
do{
     next = open.remove(0);
     //lê vizinhos do nodo atual;
     Collection<EntityID> neighbours = graph.get(next);
     if (neighbours.isEmpty()) {
        found = false;
        break;
     }
     reject = new LinkedList<EntityID>();
     divisor = calcDivisor(neighbours, model);
     //cálculo para a escolha do vizinho
     if(!found && divisor > 0){
         //calcula probabilidade individual.
         probs = calcProbIndividual(neighbours, model, divisor);
         Prob escolhida = escolheRua(probs);
         if(escolhida!=null){
            EntityID escolha = escolhida.getNode();
            open.add(escolha);
            ancestors.put(escolha, next);
            prev.add(next);
            distância += calcDistance(model,escolha, next);
            next = escolha;
         }else{
            found = true;
         }
     }
     if(found || divisor == 0){
         EntityID ultimo = defineUltimoNodo(neighbours, model);
         open.add(ultimo);
          ancestors.put(ultimo, next);
         prev.add(next);
         next = ultimo;
         found = true;
         break;
     }
     if(distância >= limite){
         found = true;
         break;
     }
}while (!found && !open.isEmpty());
```

Quadro 23 – Método da busca de caminho por feromônio

Nesta iteração, o nó contido na lista `open` é atribuído a variável `next`, então são lidos os nós vizinhos a ele através de um mapa definido na instancia da classe que contém todas as ruas e seus vizinhos, inclusive as construções. Se não existirem vizinhos o método é

encerrado.

Para o algoritmo de busca foi definido que seriam utilizadas somente as ruas para definir o caminho, pois é nelas que estão depositados os feromônios. As construções foram excluídas do algoritmo de busca sendo usadas somente em último caso para serem o ultimo nó do caminho. Para armazenar os nós vizinhos rejeitados, no caso as construções, foi definida a variável `reject`.

Com os vizinhos lidos, é calculado o divisor da fórmula definida no Quadro 3 para o cálculo da probabilidade de escolha do caminho. O método responsável por esta parte é o `calcDivisor`, que tem como entrada a lista com os vizinhos e o modelo do mundo no *time-step* atual para ler o feromônio contido nas ruas. Neste método também são adicionadas as construções a variável `reject`.

Após o divisor da fórmula ser calculado, se tiver valor maior que zero é calculada a probabilidade para cada rua vizinha ao nó atual. Este cálculo é feito através do método `calcProbIndividual`, que aplica a fórmula definida no Quadro 3. Como entrada o método recebe os vizinhos do nó atual, o modelo do mundo e o divisor calculado anteriormente. Este método retorna uma lista com as probabilidades de cada rua calculadas.

As variáveis necessárias para o cálculo da fórmula para a definição do caminho são definidas no arquivo de configuração dos agentes (`pheror-agent.cfg`) conforme mostra o Quadro 24.

```
1.  !include common.cfg
2.  search.const_k:0
3.  search.const_n:2
4.  search.const_limite:80000
```

Quadro 24 – Arquivo de configuração dos agentes

Na linha 1 é incluído o arquivo com as configurações de conexão com o *kernel*. A variável *k* é definida com valor 0 pelo parâmetro `search.const_k`, a variável *n* é definida com valor 2 pelo parâmetro `search.const_n` e o limite com valor 80000 pelo parâmetro `search.const_limite`.

Para escolher o caminho a partir das probabilidades calculadas foi definido o método `escolheRua` que sorteia a rua conforme um número aleatório gerado na execução do método. A rua com a maior probabilidade tem maior chance de estar na faixa de valor do número aleatório e assim ser escolhida pelo método.

Caso haja algum problema na busca por caminhos, seja por falta de vizinhos ao nodo atual ou por somente haverem construções como vizinhos, o divisor será igual a zero e a probabilidade não será calculada. Então será chamado o método `defineUltimoNodo` que

definirá uma construção como nó final do caminho. Caso não haja mais nenhum nó o algoritmo para e o caminho é montado com a estrutura obtida até então.

3.4 RESULTADOS E DISCUSSÃO

Para avaliar a implementação foram realizados experimentos utilizando o mapa Kobe4, (SOURCE FOURGE, 2003), usado na última RCR. Os experimentos utilizam apenas agentes bombeiros e simulação de incêndio. Portanto foram desativados os simuladores `ClearSimulator` e `CollapseSimulator` responsáveis pelos bloqueios para que o agente pudesse se locomover livremente pelo mapa. A simulação também não possui civis a serem resgatados, já que não haverá agentes do time de ambulância eles não tem função. Foram usadas quantidades diferentes de bombeiros para os cenários. Primeiro foram feitos testes com um time de 20 agentes, conforme apresenta a Figura 16. Depois foram feitos testes com um time com 40 agentes. Os 20 agentes adicionais citados tiveram como posição inicial a mesma dos outros 20 da primeira parte da simulação.

Figura 16 – Imagem com a posição inicial dos agentes, refúgios e incêndios

A implementação de agente utilizada nos experimentos é a do `SwarmFireBrigade`,

que foi definida na operacionalidade. Os experimentos realizados adotam os valores das constantes 0 para *k*, 2 para *n e* 80000 para o limite. Foi definido este valor para o limite por representar, após testes, uma distância aceitável para o deslocamento do agente dentro de um *time-step*.

Para comprovar a utilização do feromônio presente nas ruas para a escolha de caminho pelos agentes `SwarmFireBrigade`, o Quadro 25 apresenta um trecho de *log* da execução de um agente. O trecho relata a execução do método de busca de caminho onde são calculadas as probabilidades de cada rua, e depois a rua é escolhida de forma aleatória tendo maior peso de escolha a rua com maior probabilidade.

```
1. Inicio Busca.
2. Nó Inicial: 266
3. Iterando sobre os vizinhos de: 266: [279, 269]
4. Nós que já passou: []
5. Cálculo do Divisor:
6. Rua: 279 -> Feromônio: 0.5011297878809273
7. Rua: 269 -> Feromônio: 0.0
8. Divisor Calculado: 0.2511310643015832
9. Nós Rejeitados: []
10.Probabilidade individual (Para cada Nó):
11.Probabilidade Rua: 279: 1.0
12.Probabilidade Rua: 269: 7.819173592005166E-90
13.Lista probabilidades de cada nó:
14.[[ 279 ; 1.0 ], [ 269 ; 7.819173592005166E-90 ]]
15.Aleatório (determina a escolha): 0.03452323283660941
16.Contador: 0.0
17.Escolheu o nodo - > Road (279)
18.Distância até a rua 279 : 20572.196868589413
19.Iterando sobre os vizinhos de: 279: [971, 274, 266]
20.Nós que já passou: [266]
21.Cálculo do Divisor:
22.Rua: 971 -> Feromônio: 0.3006778727285564
23.Rua: 274 -> Feromônio: 0.5011297878809273
24.Road (266) - > Não é caminho elegível
25.Divisor Calculado: 0.34153824745015315
26.Nós Rejeitados: [266]
27.Probabilidade individual ( Para cada Nó ) :
28.Probabilidade Rua: 971: 0.26470588235294124
29.Probabilidade Rua: 274: 0.7352941176470589
30.Lista probabilidades de cada nó:
31.[[ 971 ; 0.26470588235294124 ], [ 274 ; 0.7352941176470589 ]]
32.Aleatório (determina a escolha): 0.9145238450501878
33.Contador: 0.0
34.Rua 971 não caiu na prob
35.Contador: 0.26470588235294124
36.Escolheu o nodo - > Road (274)
37.Distância até a rua 274 : 31234.779295000008
38.....
```

Quadro 25 – Trecho de *log* demonstrando o uso de comunicação pelo ambiente na busca de caminho

O ciclo do método de escolha de caminho, definido no Quadro 22, pode ser visto das linhas 1 a 18 do Quadro 25. Nelas é possível ver a iteração sobre os vizinhos ao nó exibido na linha 2, a leitura do feromônio contido nelas (linhas 6 e 7) o divisor calculado na linha 8 e a

probabilidade calculada nas linhas 11 e 12.

A escolha do nó 279, apontada na linha 17, ainda no Quadro 25, se deve ao fato de esta rua conter maior quantidade de feromônio do que a outra rua. Assim ela teve maior probabilidade de ser escolhida, como mostrado nas linhas 11 e 12. Na linha 18 é mostrada a distância calculada do ponto inicial (rua 266) até o ponto escolhido (rua 279).

A partir da linha 18 o ciclo de iteração é recomeçado a partir da rua 279, escolhida anteriormente. É iterado sobre os vizinhos do nó (linha 19), então a quantidade de feromônio presente em cada rua é lida e o divisor da fórmula é calculado (linhas 21 a 23). Na linha 24 é apontado como nó não elegível a rua 266. Isto acontece por este nó já fazer parte do caminho, evitando assim que seja definido um caminho circular, por esta razão este nó é rejeitado na linha 26. Depois disto as probabilidades de cada rua são calculadas (linhas 27 a 31) e a rua é escolhida de acordo com o número aleatório (linhas 32 a 36). Com a rua escolhida é calculada a distância do caminho até ela (linha 37).

A escolha de caminho segue iterando sobre os vizinhos até que seja alcançada a distância limite estipulada em configuração. O Quadro 26 apresenta outro trecho de *log* da execução do agente, onde é possível verificar a finalização da busca do caminho.

```
1.  Iterando sobre os vizinhos de: 274: [255, 279, 976]
2.  Nós que já passou: [266, 279]
3.  Cálculo do Divisor:
4.  Refuge (255) - > Não é caminho elegível
5.  Road (279) - > Não é caminho elegível
6.  Rua: 976 -> Feromônio:0.0
7.  Divisor Cálculo: 1.9636373861190906E-90
8.  Nós Rejeitados: [255, 279]
9.  Probabilidade individual ( Para cada Nó ) :
10. Probabilidade: 1.0
11. Lista probabilidades de cada nó: [[ 976 ; 1.0 ]]
12. Aleatório (determina a escolha): 0.27535374351500463
13. Contador: 0.0
14. Escolheu o nodo - > Road (976)
15. Distância até a rua 976 : 63227.87983245964
16. Iterando sobre os vizinhos de: 976
17. [926, 927, 274, 255, 924, 925, 922, 975, 923, 921, 929, 928]
18. Nós que já passou: [266, 279, 274]
19. ...
56. Escolheu o nodo - > Road (922)
57. Distância até a rua 922 : 81320.64858132724
58. Venceu limite: 81320.64858132724
59. Resultado da busca
60. Next: 922
61. Ancestors: {274=279, 922=976, 279=266, 266=266, 976=274}
62. Caminho [279, 274, 976, 922]
63. Fim Busca.
```

Quadro 26 – Trecho de *log* demonstrando a finalização da busca de caminho

O trecho omitido do *log* (linha 19 a 55) somente ilustra o cálculo da probabilidade para as ruas vizinhas apresentadas na linha 18 do Quadro 25. Conforme mostram as linhas finais do Quadro 26, após a distância estipulada ser atingida, o método encerra a iteração sobre as

ruas e monta o caminho a ser retornado.

A Tabela 1 apresenta o resultado de várias simulações executadas com o `SwarmFireBrigade`, isto é, usando a extensão SI. A quantidade de feromônio a ser depositada foi definida inicialmente com o valor 1 (um). A coluna *Quantidade de Bombeiros* se refere ao número de agentes bombeiros utilizados nas simulações. A coluna *Percentual de Evaporação* diz respeito ao coeficiente de evaporação, mostrado na fórmula do Quadro 4, adotado para as simulações. Todos os parâmetros mostrados nas tabelas, exceto o *score*, tem seus valores definidos por meio de arquivos de configuração. O *score* obtido com as simulações, apresentado na coluna *Score*, leva em consideração somente as construções, se elas foram salvas ou o fogo as destruiu e a quantidade de dano que o fogo causou às construções, conforme apresentado no Quadro 1. O fato de o *score* ser maior significa que o time de agentes conseguiu salvar uma área maior de construções no mapa, sendo assim melhor a pontuação mais alta. Este *score* é uma média calculada de três resultados obtidos para cada experimento.

Tabela 1 – Resultados das simulações com os agentes *SwarmFireBrigade* depositando 1 (uma) unidade de feromônio

Quantidade de Bombeiros	**Percentual de Evaporação**	***Score***
20	0%	0,205543138
20	25%	0,196124823
20	50%	0,206619558
20	**75%**	**0,223827752**
20	100%	0,193934737
40	**0%**	**0,356805741**
40	25%	0,263814039
40	50%	0,286295404
40	75%	0,269387903
40	100%	0,258381419

Para efeito de comparação, foram executados experimentos no mesmo mapa, e com as mesmas quantidades de agentes, mas com um time de agentes que não utiliza a marcação de caminho por feromônio. A estratégia destes agentes é a de, se estiver com água, procura por incêndio próximo, se estiver sem água vai até o refúgio. Se estiver com água e achar um incêndio, vai apagar o incêndio através do trajeto mais curto. Se não achar o incêndio vaga pelo mapa aleatoriamente até encontrar um incêndio. Este agente é distribuído junto com o código fonte do simulador, e chama-se *SampleFireBrigade*. A Tabela 2 apresenta os resultados obtidos com o *SampleFireBrigade*.

Tabela 2 – Resultados Resultados das simulações com os agentes *SampleFireBrigade*

Quantidade de Bombeiros	***Score***
20	0,226364196
40	0,239583061

Mediante os resultados apresentados observa-se que com o depósito de 1 unidade de feromônio para um time de 20 agentes *SwarmFireBrigade* (com comunicação pelo ambiente) os resultados obtidos se aproximam com os de um time de mesmo número de agentes *SampleAgent* (sem comunicação pelo ambiente). Isto se deve ao fato de que as ruas, no inicio da simulação, não estavam marcadas com feromônios indicando a direção dos incêndios. Assim os agentes vagaram de forma aleatória até encontrarem um incêndio e o apagarem.

Com relação aos experimentos realizados com 40 agentes *SwarmFireBrigade*, os resultados obtidos demonstram que houve maior *score* em relação ao time de agentes *SampleFireBrigade*. O fato pode ser justificado devido a existência de maior número de agentes no mapa agindo na tarefa de combater os incêndios, levando a maior depósito de feromônio nas ruas devido à movimentação destes agentes em direção aos refúgios. Assim o caminho para os incêndios permanece marcado por mais tempo e sua marcação tem maior influência na decisão dos demais agentes.

Posteriormente foram efetuados testes com outros cenários em relação ao depósito de feromônios pelos agentes *SwarmFireBrigade*. Primeiramente com 5 unidades de feromônio, tendo os resultados apresentados na Tabela 3.

Tabela 3 – Resultados das simulações com os agentes *SwarmFireBrigade* depositando 5 (cinco) unidades de feromônio

Quantidade de Bombeiros	**Percentual de Evaporação**	***Score***
20	0%	0,206738148
20	**25%**	**0,217581256**
20	50%	0,213188776
20	75%	0,204559179
20	100%	0,185567313
40	0%	0,236505086
40	25%	0,275826638
40	**50%**	**0,334192962**
40	75%	0,306718365
40	100%	0,309240701

Com estes testes ainda pode-se notar o melhor desempenho obtido pelo time com 40 agentes, mas sua pontuação geral diminuiu. A justificativa para isto ter ocorrido pode ser o fato de que com uma quantidade maior de feromônio sendo depositada, a rua continua marcada por mais tempo, podendo assim continuar marcada mesmo quando perder sua

importância. No caso, o incêndio ser apagado ou a construção completamente destruída.

Após estes testes foram efetuadas simulações com cenário onde cada agente depositaria 10 unidades de feromônios para a marcação do caminho. Os resultados são demonstrados na Tabela 4.

Tabela 4 – Resultados das simulações com os agentes *SwarmFireBrigade* depositando 10 (dez) unidades de feromônio

Quantidade de Bombeiros	**Percentual de Evaporação**	***Score***
20	0%	0,203687695
20	25%	0,190899442
20	50%	0,212238217
20	75%	0,195587738
20	**100%**	**0,218182847**
40	0%	0,317462211
40	25%	0,284331181
40	50%	0,323702942
40	**75%**	**0,355580613**
40	100%	0,262838562

O time com 40 agentes ainda tem seu desempenho superior em relação ao time com somente 20 agentes. Com 10 unidades de feromônio sendo depositadas por cada agente na rua, a probabilidade de escolha é elevada rapidamente. Assim, as ruas marcadas terão maior probabilidade de serem escolhidas em muito menos tempo.

Com os testes realizados pôde-se construir um comparativo dos melhores resultados entre obtidos em todos os experimentos, conforme mostra o Quadro 27.

Sampefirebrigade	**1 (uma) unidade de feromônio depositado**	**5 (cinco) unidades de feromônio depositado**	**10 (dez) unidades de feromônio depositado**
40 agentes	40 agentes	40 agentes	40 agentes
sem evaporação	0% de evaporação	50% de evaporação	75% de evaporação
0,239583061	0,356805741	0,334192962	0,355580613

Quadro 27 – Comparativos entre os melhores resultados

Com isto observou-se que o time de 40 agentes obteve maior pontuação em todos os experimentos. Isto se deve ao fato de que há um maior número de agentes percorrendo o mapa durante a simulação.

Ainda observou-se que o coeficiente de evaporação tem relação com a quantidade de feromônio depositado pelos agentes. Quanto maior a quantidade depositada, maior foi necessário ser o coeficiente de evaporação para que o desempenho dos agentes não diminuísse. Como há maior quantidade de feromônio sendo depositada nas ruas pelos agentes a cada ciclo, é necessário que uma maior quantidade de feromônio seja evaporada para que um caminho que não tem mais importância deixe de ter influencia na escolha pelo agente

durante a simulação.

Todos os testes foram executados com o cálculo de evaporação sobre a quantidade total de feromônio presente na rua e não sobre cada quantidade depositada em separado.

Com relação aos trabalhos correlatos, no algoritmo definido por Kassabalidis et al. (2001) é utilizado o conceito de SI em relação à definição de rotas, para se encontrar a melhor rota para o tráfego de rede, dispensando o uso de troca de mensagens para determinar as tabelas de roteamento. Porém, não são determinados quais agentes específicos para explorar a rede e os nós, nem quais irão montar a tabela, já que tudo é feito através de envio de pacotes e cálculo de atraso. Já o trabalho de Santos (2009) sugere que seja desenvolvido uma solução que substitua a comunicação direta entre os agentes, utilizando assim a comunicação através do ambiente. Neste sentido, a extensão SI desenvolvida neste trabalho atende a sugestão feita por Santos (2009).

4 CONCLUSÕES

O simulador RCR tem como seu objetivo realizar competições para o aprimoramento de estudos na área da IA e de desenvolvimento de SMA´s. No entanto, o simulador não oferecia nenhum recurso em SI tendo somente como opção de comunicação a troca de mensagens através de rádio. Para eliminar esta limitação, este trabalho desenvolveu uma extensão SI, que permite desenvolver agentes que utilizam comunicação pelo ambiente. Os experimentos realizados em um time de agentes que utiliza comunicação pelo ambiente demonstraram que seu uso é promissor, proporcionando melhor *score* do que times de agentes que não utilizam comunicação pelo ambiente.

Durante o trabalho foi desenvolvido um estudo sobre as áreas de IA e SMA, principais conceitos de SI especialmente no que tange a comunicação através do ambiente, neste caso feromônios. Também foi realizado um profundo estudo no funcionamento e estrutura do simulador RCR. Como o simulador RCR é um projeto de dimensões muito complexas levou-se certo tempo para o entendimento de seu funcionamento e operação, pois não há documentação disponível referente ao seu projeto e arquitetura. Este fator constituiu a principal dificuldade encontrada durante a realização deste trabalho, consumindo a maior parte do tempo. Como conseqüência, não foi possível realizar o desenvolvimento dos agentes policiais e ambulâncias para testes de operacionalidade.

Como foi necessário alterar classes existentes do simulador, um usuário que pretende desenvolver um agente utilizando a extensão, precisa ter o código do simulador alterado, e não apenas o código do simulador disponível para download. Outra limitação é em relação à quantidade de experimentos realizados que não abrangeram outros possíveis cenários com diferentes configurações para análise, como quantidade diferente para o depósito de feromônios e quantidades de agentes no mapa.

Com este trabalho pode-se concluir também que o desenvolvimento de uma extensão para disponibilizar recursos de SI no simulador RCR é viável, porém torna-se complexa quando é preciso apenas adicionar códigos ao simulador sem alterar suas funcionalidades já presentes.

4.1 EXTENSÕES

Como extensão para o presente trabalho propõe-se:

a) estudo e adaptação do algoritmo eXtreme-Ants de Santos (2009) para a utilização dos recurso de SI disponibilizados pela extensão. Para isto será necessário estudar e adaptar o time de agentes implementado por Santos para utilizarem a comunicação pelo ambiente disponibilizada pela extensão.
b) análise e implementação de outros agentes utilizando os recursos oferecidos pela extensão. Será necessário definir, de acordo com cada tipo de agente, qual a melhor estratégia de comunicação pelo ambiente poderá ser usada.
c) modificação do método de gravação e leitura da quantidade de feromônio depositado para que seja possível depositar e evaporar em separado cada deposito de feromônio feito pelo agente e comparar com a atual extensão. A forma atual compreende o armazenamento em um único atributo com todo o feromônio depositado. Sugere-se que seja altera para uma estrutura onde seja possível armazenar cada quantidade de feromônio depositada na rua e calcular a evaporação para cada uma destas quantidades. Com isto ainda deverá ser permitido ao agente somente ler a quantidade total do feromônio presente na rua. Após isto, comparar se houve melhora no desempenho dos agentes em relação ao *score* obtido.

REFERÊNCIAS BIBLIOGRÁFICAS

BARR, Avron; FEIGENBAUM, Edward A. **The handbook of artificial intelligence**. [S.l.]: Addison-Wesley, 1982. 428 p.

BONABEAU, Eric; THERAULAZ, Guy; DORIGO, Marco. **Swarm intelligence**: from natural to artificial systems. New York: Oxford University Press, 1999. 307 p.

DENEUBOURG, Jean-L et al. The self-organizing exploratory pattern of the argentine ant. **Journal of Insect Behavior**, United States, v. 3, p. 159-168, 1990.

ECLIPSE. [S.l.], 2008. Disponível em: <http://www.eclipse.org>. Acesso em: 10 fev. 2011

JENNINGS, Nicholas R.; SYCARA, Katia; WOOLDRIGE, Michael J. A roadmap of agent research and development. **Autonomous Agents and Multi-Agent Systems**, [S.l.], n. 1, p. 275–306, 1998.

KASSABALIDIS, Ikas et al. Swarm intelligence for routing in communication networks. In: GLOBAL TELECOMMUNICATIONS CONFERENCE, 2. , 2001, San Antonio. **Proceedings...** Seattle: Washigton University, 2001. p. 3613–3617. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.8398&rep=rep1&type=pdf>. Acesso em: 27 mar. 2011.

KITANO, Hiroaki; TADOKORO, Satoshi. RoboCup rescue, a grand challenge for multiagent and intelligent systems. **AI Magazine,** [S.l.], v. 22, n. 1, p. 39-52, 2001. Disponível em: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/1542/1441>. Acesso em: 27 mar. 2011.

LARSEN, Flemming N. **Robocode**: build the best – destroy the rest! [S.l.], [2010]. Disponível em: <http://robocode.sourceforge.net/>. Acesso em: 17 mar. 2011.

ROBOCUP. **RoboCup soccer**. Osaka, [2010]. Disponível em: <http://www.robocup.org/robocup-soccer/>. Acesso em: 17 mar. 2011.

ROBOCUP RESCUE. Tokio, 2006. Disponível em: <http://www.robocuprescue.org/>. Acesso em: 17 mar. 2011.

ROBOCUP RESCUE SIMULATION LEAGUE. **Rules and setup**. [S.l.], 2010. 21 p. Disponível em<http://www.aaai.org/AITopics/assets/PDF/AIMag19-02-2-article.pdf>. Acesso em: 24 maio 2011.

SANTOS, Fernando. **eXtreme-Ants**: algoritmo inspirado em formigas para alocação de tarefas em extreme teams. 2009. 69 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SESAM. **Multi-agent simulation environment.** Germany, [2010]. Disponível em: <http://www.simsesam.de/>. Acesso em: 17 mar. 2011.

SOURCE FORGE. **Robocup rescue simulation project**. [S.l.], 2003. Disponível em: <http://sourceforge.net/projects/roborescue/>. Acesso em: 18 fev. 2011.

SPARXSYSTEMS. Creswick, 2000. Disponível em: <http://www.sparxsystems.com/>. Acesso em: 15 mar. 2011.

SYCARA, Katia P. Multiagents Systems. **AI Magazine**, California, v.19, n.2, p.79–92, 1998. Disponível em: <http://www.aaai.org/AITopics/assets/PDF/AIMag19-02-2-article.pdf>. Acesso em: 15 fev. 2011.

SKINNER, Cameron; BARLEY, Mike. Robocup rescue simulation competition: status report. In: BREDENFELD, Ansgar et al. (Ed.). **RoboCup 2005**: robot soccer world cup IX. Berlin: Springer-Verlag, 2006. p. 632-639.

WOOLDRIDGE, Michael J. **An introduction to multiagent systems**. New York: John Wiley. 348p.