UMA ANÁLISE COMPARATIVA ENTRE AMBIENTES DE SIMULAÇÃO E LINGUAGENS DE PROPÓSITO GERAL PARA O DESENVOLVIMENTO DE SIMULAÇÕES.

Trabalho de Conclusão de Curso

Engenharia da Computação

Frédick Silva Uchôa Júnior
Orientador: Prof. Maria Lencastre M. Cruz

Recife, 4 de Julho de 2006

UMA ANÁLISE COMPARATIVA ENTRE AMBIENTES DE SIMULAÇÃO E LINGUAGENS DE PROPÓSITO GERAL PARA O DESENVOLVIMENTO DE SIMULAÇÕES.

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Frédick Silva Uchôa Júnior
Orientador: Prof. Dr. Maria Lencastre M. Cruz

Recife, 4 Julho de 2006

Frédick Silva Uchôa Júnior

UMA ANÁLISE COMPARATIVA ENTRE AMBIENTES DE SIMULAÇÃO E LINGUAGENS DE PROPÓSITO GERAL PARA O DESENVOLVIMENTO DE SIMULAÇÕES.

Resumo

Este trabalho apresenta uma análise comparativa entre um ambiente de simulação genérico (Arena) e uma linguagem de propósito geral (Java) no desenvolvimento de um mesmo estudo de caso. A simulação vem conquistando uma importância cada vez maior, sendo utilizada em várias áreas tais como: linhas de produção, logística, transporte (por exemplo, aéreo, ferroviário, rodoviário), comunicação e entre outras. Portanto é muito importante escolher a ferramenta adequada para o desenvolvimento de um modelo, visto que cada ferramenta possui suas vantagens e desvantagens. A simulação inicialmente utilizou ferramentas de linguagens de propósito geral (que fornecem muita liberdade e dificuldade em sua utilização). Mas, à medida que os recursos computacionais cresceram surgiram os ambientes de simulação genérico, que possibilitaram várias facilidades. Mas por outro lado implica em uma certa perda de liberdade na modelagem. Nesse contexto o presente trabalho também traz um estudo histórico das ferramentas de simulação. Este trabalho descreve um estudo de caso baseado num sistema de tráfego de automóveis, sendo esta escolha em virtude da riqueza de informações que podem ser extraídas neste exemplo.

Abstract

This work presents a comparative analysis between a generic simulation framework (Arena) and a general purpose language (Java) in the development of a same study of case. The simulation is conquering each time more importance, being used in several areas such production as: production lines, logistics, transports (for instance, aerial, rail, road), communication and among others. Therefore it's very important to choose the appropriate tool for the development of a model, because each tool possesses advantages and disadvantages its use. The simulation initially was used tools of on general purpose languages (that supply a lot of freedom and difficult its use). But, as the resources computation increased the generic simulation frameworks also appear. But on the other hand it implicates in a certain loss of freedom in the modeling. In that context the present works also brings a historical study of the simulation tools. This work describes a study of cases based in a system of automobile traffic, this choice because of the wealth of information that they can be extracted in this example.

Sumário

Índice de Figuras **iv**

Índice de Tabelas **v**

1 Introdução **7**

Objetivo 8
Organização do Texto 9

Simulação **10**

2.1 Considerações iniciais 10
2.2 Vantagens e Desvantagens da Simulação 11
2.3 Sistemas e Modelos 12
2.4 Conceitos Básicos da Modelagem e Simulação 14
2.5 Ferramentas de Simulação 15
2.5.1 Evolução das linguagens de simulação 15
2.5.2 Linguagens de propósito geral 16
2.5.3 Ambientes de Simulação 17
2.5.4 O ambiente de Desenvolvimento Arena 18
2.6 Procedimento para um estudo de Modelagem e Simulação 20
2.7 Simulação Distribuída 22
2.8 Simulação na *web* 24
2.9 Arquitetura de Alto Nível para Simulação 24

Estudo de Caso **27**

3.1 Descobrimento e Orientação 27
3.1.1 Formulação do Problema 27
3.2 Construção do Modelo e Coleta de Dados 29
3.2.1 Concepção do Modelo de Tráfego 29
3.2.2 Coleta de Dados 31
3.2.3 Tradução do Modelo 32
3.2.3.1 Tradução do modelo no Arena 32
3.2.3.2 Tradução para modelo em Java 38
3.2.4 Verificação e Validação 44
3.3 Execução do Modelo 44
3.3.1 Projeto Experimental e Análise dos Resultados 44
3.5 Considerações finais 47

Conclusões e Propostas para Trabalhos Futuros **49**

4.1 Conclusões 49
4.2 Trabalhos Futuros 50

Índice de Figuras

Figura 1. Representação de um modelo de sistema. 12
Figura 2. Formas de estudo de um sistema. 13
Figura 3. Modelagem com uma linguagem de propósito geral. 17
Figura 4. Modelagem com um ambiente de simulação. 18
Figura 5. Passos para modelar um problema utilizando modelagem e simulação. 21
Figura. 6. Classificação dos protocolos de simulação distribuída. 23
Figura 7. Sistema simulado. 28
Figura 8. Passos de mudança de cor de um semáforo. 28
Figura 9. Módulo *Create* do Arena. 32
Figura 10. Gráfico de ritmo de chegada. 33
Figura 11. Configuração do bloco inicialização. 34
Figura 12. Representação da estrada. 34
Figura 13. Funcionamento do semáforo. 35
Figura 14. Atribuição de variáveis na mudança de cores. 36
Figura 15. Verifica fila e semáforo. 36
Figura 16. Módulo D*ispose*. 36
Figura 17. Parte da Estrada. 1 37
Figura 18. Diagrama de Classes da simulação. 38
Figura 19. Os atributos e os principais métodos da classe Carro. 39
Figura 20. Diagrama de Atividades do Carro. 39
Figura 21. Atributos e parâmetros da classe Estrada. 40
Figura 22. Fluxograma para atualizar_lista_de_objetos_na_estrada(). 40
Figura 23. Os atributos e principais métodos do semáforo. 41
Figura 24. Classe TesteDaSimuação. 42
Figura 25. Trecho de código de criação de três estradas, dois semáforos e conexão das estradas. 42
Figura 26. Tempo de simulação e velocidade de execução. 42
Figura 27. Trecho de código do método criarVeículo(). 43
Figura 28. Código da atualização dos objetos. 43

Índice de Tabelas

Tabela 1. Histórico do uso da simulação computacional. 15
Tabela 2. Comparação das vantagens e desvantagens das linguagens. 19
Tabela 3. Intervalo de chegada da Estrada 1. 31
Tabela 4. Intervalo de chegada da Estrada 2. 31
Tabela 5. Intervalo de chegada da Estrada 3. 31
Tabela 6. Módulos utilizados na simulação. 37
Tabela 7. Parâmetros estudados no modelo. 45
Tabela 8. Resultados dos testes no Arena. 45
Tabela 9. Resultados dos testes no Java. 46
Tabela 10. Análise do sistema implementado. 47
Tabela 11. Atributo de Qualidade. 48
Tabela 12. Lista dos Modelos simulados e seus parâmetros. 78
Tabela 13. Resultado completo dos testes no Arena. 79
Tabela 14. Resultado completo dos testes em Java. 79

Agradecimentos

Agradeço primeiramente a DEUS que me deu força e perseverança para concluir este trabalho. Agradeço especialmente a minha mãe e titia que se esforçaram ao máximo para que eu pudesse entrar, permanecer e concluir curso de graduação. Agradeço a todos os meus familiares e minha namorada por toda compreensão, amor e dedicação que tiveram comigo. Agradeço de todo coração ao grupo de oração Família Renascer que não cessaram de interceder por mim junto a Jesus durante todos momentos deste trabalho.

Por fim agradeço especialmente a professora Maria Lencastre, orientadora deste trabalho, por toda atenção durante a composição desta monografia, ao professor Carlos Alexandre que sempre deu atenção para todos os alunos e a todo corpo docente da faculdade.

Capítulo 1

Introdução

A simulação computacional de sistemas, ou simplesmente simulação [5], é uma técnica que tem sido bastante utilizada nos últimos tempos por profissionais de diversas áreas. Essa técnica ajuda-os a responder questões do tipo: "o que aconteceria se?". Sua utilização vem tornado-se cada vez mais comum, devido ao seu poder de investigar sistemas. Sem que os mesmos sofram perturbação, de estudar "gargalos" ou pontos críticos em sistemas complexos do mundo real. Além de poder estudar sistemas que ainda não existem, levando ao desenvolvimento de projetos mais eficientes. Tudo isso atrelado a um menor custo, uma vez que estes estudos são realizados no computador.

Hoje em dia, impulsionados pelas necessidades do mundo real e pelo avanço das tecnologias, os sistemas de simulação se apresentam bem mais complexos do que os seus predecessores. A simulação se mostrou uma ferramenta estratégica para estudos de reengenharia, mudaças de *layout*, planejamento de produção, logística, sistemas de controle de tráfego, sistemas de gerenciamento de batalhas em tempo real, sistema de resposta de emergência, etc.

Existem diferentes abordagens para desenvolvimento desses sistemas de simulação. Entre os diferentes tipos de aborgagems temos as linguagens de propósito geral, as bibliotecas pré-construídas, as linguagens de simulação, os programas geradores e os ambientes de simulação genérico. Cada uma dessa abordagem atende a uma certa necessidade, pois em alguns momentos é preciso ter mais liberdade e controle da simulação, em outros é preciso ter mais facilidades para compor o modelo.

O crescente interesse em usar a *web*, como uma nova plataforma, também afetou a àrea de simulação. J. Kuljis et al [10]. Destaca que existe uma grande variedade de ambientes e linguagens para a simulação baseada na *web*, tomando como base a linguagem Java. Existe um conjunto de aplicações que podem usufruir bastante das vantagens da *web*, tais como: aquelas que lidam com grande quantidade de dados, e aquelas que permitem que usuários em múltiplos *sites* colaborem em um projeto de um modelo.

Por outro lado, o avanço da complexidade dos sistemas computacionais trouxe um aumento significativo no tempo de execução da simulação seqüencial, tornando seu tempo de processamento muitas vezes inviável, principalmente para a obtenção de uma grande quantidade de resultados. Este avanço motivou a adoção da simulação distribuída, que por sua vez tem como principal objetivo reduzir o tempo de processamento de um programa de simulação.

Nos anos noventa, o Departamento de defesa dos Estados Unidos numa ação conjunta com o governo, a academia e a indústria desenvolveram o padrão HLA (*High Level Architecture*), uma arquitetura de software de alto nível que permite combinar simulações de

computador em uma simulação maior [20]. O HLA é baseado em objeto e componentes, e trata problemas de integração, compartilhamento de dados, comunicação, sincronização e gerenciamento de tempo.

As pesquisas na área de simulação distribuída vêm crescendo nos últimos vinte anos [6], principalmente com o surgimento de novas áreas de aplicação. Pode-se citar, como exemplo, controle de tráfego aéreo, treinamentos militares promovidos através de simulação, simuladores de vôos, sistemas de transporte, simuladores de jogos de guerra [14] [21], simulação de redes de comunicação e de computadores [7], além de pesquisas em laboratórios industriais e em universidades por todo o mundo. Em conseqüência, surgiram protocolos para garantir a sincronização entre os processos da simulação distribuída e para permitir sua execução de maneira correta.

A determinação da ferramenta adequada para um problema é uma tarefa complexa. Uma escolha inadequada pode comprometer o desenvolvimento da simulação, encarecendo o projeto e aumentando o tempo de modelagem. Portanto, quanto mais se souber do problema e das características de cada ferramenta mais adequada será a escolha.

Ao longo deste trabalho podemos pode-se observar que a técnica de simulação permite imitar o funcionamento de um sistema real no computador, com diferentes níveis de profundidade, possibilitando a análise de diversas alterações no cenário virtual, sem o custo e risco de atuar no cenário real. Além do mais, verifica-se que a simulação é utilizada no estudo de diversos tipos de sistemas, e está se desenvolvendo em diversas áreas, sendo uma técnica bastante promissora.

Objetivo

Esta monografia visa fundamentalmente fazer uma comparação entre duas ferramentas de simulação, especificamente o Arena [19], ambiente de simulação genérico, e Java [22] (linguagem de programação de propósito geral); além de levar a um entendimento a respeito de suas características, sua importância, bem como as dificuldades encontradas em cada abordagem.

Neste trabalho também será feita implementação de um estudo de caso, o qual permitirá explorar os conceitos de simulação abordados ao longo do texto, levando a um entendimento mais consistente dos mesmos.

Este trabalho tem como objetivo a realização de uma análise comparativa entre duas abordagens que podem ser usadas para desenvolver uma simulação. O trabalho analisa um ambiente de simulação genérico e uma linguagem de programação de propósito geral. Ao longo do trabalho é implementado um estudo de caso, baseado num sistema de tráfego, no qual todas as entidades envolvidas (carro, estrada, semáforo, dados de entrada, etc) serão hipotéticos. A partir deste objetivo podemos delimitar alguns objetivos específicos, os quais irão guiar o processo de desenvolvimento deste trabalho.

- Identificar e reconhecer a classificação dos ambientes de simulação.

- Estudar o crescente interesse em usar a *web* como uma nova plataforma para simulação.

- Identificar que a simulação apresenta um custo associado à computação, ou seja, quando se deseja simular um sistema mais complexo o tempo de simulação pode se tornar inviável.

- Identificar que a simulação distribuída apresenta um custo associado à comunicação, especificamente na sincronização das mensagens, por causa da necessidade de criação de protocolos de sincronização.

- Justificar a utilização de simulação, descrevendo suas principais peculiaridades.

- Modelar o mesmo estudo de caso, por exemplo: as entidades (semáforo, carro e a estrada), os eventos (chega de carro, mudança de cor do semáforo, etc) e outras características do sistema, no Arena e no Java.

- Simular o modelo, e fazer uma comparação entre as duas implementações quanto ao tempo de desenvolvimento, dificuldades, tempo de processamento, validade dos resultados, etc.

- Analisar a eficiência da escolha correta da ferramenta de simulação abordada, observando-se os resultados obtidos.

Organização do Texto

Este trabalho se encontra estruturado conforme descrito a seguir:

Capitulo 2 – Neste capítulo são abordados os principais tópicos referentes à simulação computacional de sistemas. Em especial, será abordada a motivação do surgimento da simulação, as vantagens e desvantagens da simulação, os modelos de simulação e suas classificações. O capítulo também aborda a história das linguagens de simulação, e apresenta um processo para o desenvolvimento de uma simulação. Por fim, apresenta aspectos relacionados com a simulação distribuída, a simulação na *web* e a arquitetura de alto nível definida para a interoperabilidade entre simulações, também chamada HLA.

Capitulo 3 – É feita a descrição e o desenvolvimento do estudo de caso baseado nos passos para o desenvolvimento de uma simulação visto no capítulo dois. Inicialmente o problema real é apresentado, em seguida mostra-se o modelo conceitual proposto e as traduções para as duas abordagens usadas na implementação. Neste capítulo também se faz uma comparação entre ambiente de simulação Arena e a linguagem Java como possíveis abordagens para se desenvolver simulações.

Capitulo 4 – São expostas ss conclusões do trabalho e propostas para trabalhos futuros.

Capítulo 2

Simulação

2.1 Considerações iniciais

Do Aurélio: "[simulação] s.f. Ato ou efeito de Simular. Experiência ou ensaio realizado com o auxílio de modelos". Simulação é a técnica de estudar o comportamento e reações de um determinado sistema através de modelos, que imitam na totalidade ou em parte as propriedades e comportamentos deste sistema em uma escala menor, permitindo sua manipulação e estudo detalhado [5]. Estes modelos geralmente utilizam diversos parâmetros sobre a operação do sistema. Uma vez desenvolvido e validado, o modelo pode ser usado para investigar uma grande variedade de questões sobre o sistema. Mudanças no sistema podem ser simuladas a fim de prever seu impacto no desempenho do mesmo.

A simulação pode também ser usada para estudar sistemas ainda na fase de concepção, antes que sejam efetivamente implementados. Assim, a simulação pode ser usada como uma ferramenta para predizer os efeitos de uma mudança em sistemas existentes e também como uma ferramenta de projeto para avaliar e validar o desempenho de novos sistemas.

A evolução crescente da informática, nos últimos anos, tornou o computador um importante aliado da simulação de sistemas, ou simplesmente simulação. A simulação por computador é usada nas áreas mais diversas, tais como sistemas de produção, serviços militares e governamentais, serviços financeiros, serviços sociais e ambientais, etc. Costuma-se dizer que: "tudo que pode ser descrito pode ser simulado" [17] .

Dentre as técnicas disponíveis para modelagem de sistemas temos a teoria das filas e a simulação [5]. A teoria das filas é um método analítico que aborda o assunto através de fórmulas matemáticas. A simulação é uma técnica que utiliza o computador digital, para montar um modelo que melhor represente o sistema em estudo. A simulação é mais utilizada que a teoria das filas devido à sua flexibilidade, pois permite respostas rápidas com as modificações efetuadas no modelo, e ao baixo custo na solução de modelos.

Existem diferentes ferramentas para o desenvolvimento desses sistemas de simulação no computador. Cada ferramenta tem as suas vantagens e desvantagens sendo apropriada de acordo com os requisitos do modelo em estudo. Entre os diferentes tipos de abordagens temos: i) Linguagem de propósito geral, onde o usuário tem de construir todo o programa, esta abordagem é também chamada de *Do-it-yourself* (faça você mesmo). ii) Bibliotecas pré-construídas, onde rotinas são incorporadas aos programas. Compõe um complemento natural da abordagem anterior *Do-it-yourself*. iii) Linguagens de Simulação, onde a programação é feita através de uma sintaxe apropriada. iv) Programas Geradores, sistemas baseados em interfaces gráficas interativas. v)

Ambientes de Simulação Genérica, que oferecem facilidades extras para a execução da simulação baseados em modelos animados por modelos interativos de sistemas. Essas abordagens serão mais bem explicadas na sessão 2.5. O uso gráfico é complementar ao processo de modelagem. A flexibilidade é mantida através do uso de uma linguagem com uma sintaxe específica para descrição de módulos. Além do mais, J. Kuljis et al [10] destaca que existe uma grande variedade de ambientes e linguagens para a simulação baseada na web, tomando como base linguagens como Java.

Vemos que a simulação possui um grande leque de abordagens, e dentro de cada abordagem encontram-se várias ferramentas. Cada uma delas possui características próprias para programar os mais diversos tipos sistemas. A escolha correta de qual programa utilizar faz grande diferença, principalmente, no tempo de desenvolvimento, no custo do projeto, e muitas vezes na sua viabilidade como solução.

2.2 Vantagens e Desvantagens da Simulação

Em [1] [5] [15] são apresentadas algumas vantagens e desvantagens relacionadas ao uso de simulação. Dentre as vantagens, pode-se citar:

- Um modelo pode ser utilizado inúmeras vezes para avaliar projetos propostos;
- Mesmo que os dados de entrada estejam sob a forma de rascunhos, o uso da simulação permite avaliar o sistema;
- Geralmente métodos analíticos são mais difíceis de aplicar do que a simulação;
- Possuem grande flexibilidade, pois se aplicam aos mais variados modelos;
- Os modelos analíticos requerem maior número de simplificações que os modelos de simulação, que por sua vez possuem maiores níveis de detalhes que a técnica anterior, podendo assim analisar melhor o sistema;
- Hipóteses podem ser testadas e confirmadas, dependendo dos resultados;
- O tempo de simulação é independente do tempo real, ou seja, durante a simulação pode-se acelerar ou retardar a reprodução dos fenômenos, para melhor estudá-los;
- A identificação de “gargalos” fica mais fácil, principalmente com a ajuda visual, que algumas ferramentas dispõem.
- Responde questões do tipo: “o que aconteceria se?”;
- Possui um processo de modelagem evolutivo. Inicia-se com um modelo simples e aumenta-se sua complexidade aos poucos, observando as peculiaridades do problema.

- Os resultados de uma simulação, submetidos a uma série de etapas de modelagem, teste, validação e representação visual, têm melhor aceitação que a opinião de uma única pessoa.

Os mesmo autores citam também desvantagens relacionadas ao uso da simulação, dentre elas:

- Alguns sistemas complexos levam muito tempo modelando e executando. A tentativa de simplificar o modelo pode levar a resultados inconsistentes.

- Em algumas simulações os resultados são de difícil interpretação. É difícil determinar quando uma observação durante uma "rodada" da simulação se deve a alguma relação relevante do sistema, ou a processos aleatórios intrínsecos ao modelo.

- A construção de modelos requer treinamento; a técnica é aprendida e aperfeiçoada com o tempo através da experiência.

- A programação de um modelo pode tornar-se uma tarefa árdua e dispendiosa se os recursos computacionais, principalmente a ferramenta de simulação escolhida, não forem apropriados.

2.3 Sistemas e Modelos

A simulação é uma das muitas técnicas existentes para analisar sistemas. P. Freitas [5] informa que um sistema é definido como: "um conjunto de objetos, como pessoas ou máquinas, por exemplo, que atuam e interagem com a intenção de alcançar um objetivo ou um propósito lógico"

A modelagem tenta descrever e criar um modelo que imite o funcionamento do sistema real. Esse passo requer uma série de abstrações, levando a um conjunto de simplificações sobre o funcionamento do sistema modelado. A Fig 1 [5] mostra a idéia de um modelo de sistema O sistema é toda a nuvem enquanto o modelo é a parte essencial da nuvem. O modelo é uma parte do sistema real que está sendo analisado. A adição de muitos detalhes pode produzir complicações desnecessárias, e a retirada de características importantes pode invalidar o modelo [19]. Uma grande dificuldade da simulação é validar os modelos projetados, ou seja, determinar a medida da corretude do modelo e de seus resultados, isto é, o quanto se pode confiar neles.

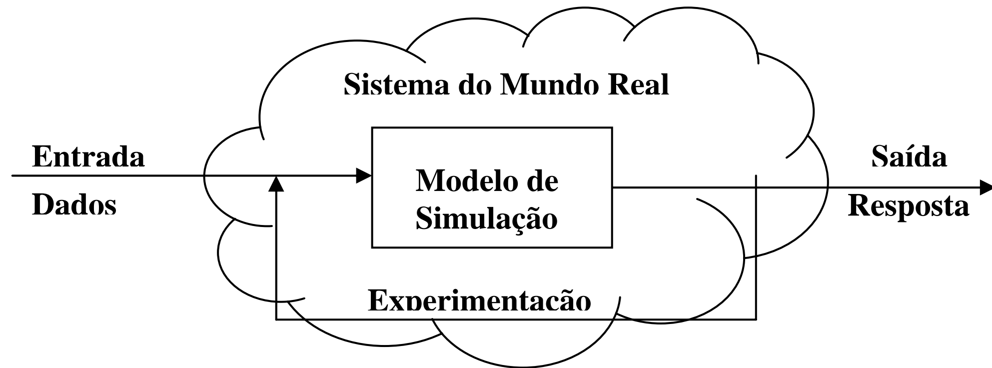

Fig 1. Representação de um modelo de sistema.

Existem vários tipos de modelos que podem ser utilizados para simular um problema. A escolha destes depende do propósito e da complexidade do sistema. Os modelos podem ser classificados como sendo matemáticos ou não matemáticos. Um modelo matemático usa notação simbólica e relações matemáticas para representar um sistema. Um modelo de simulação é um tipo particular de modelo matemático (ver Fig 2).

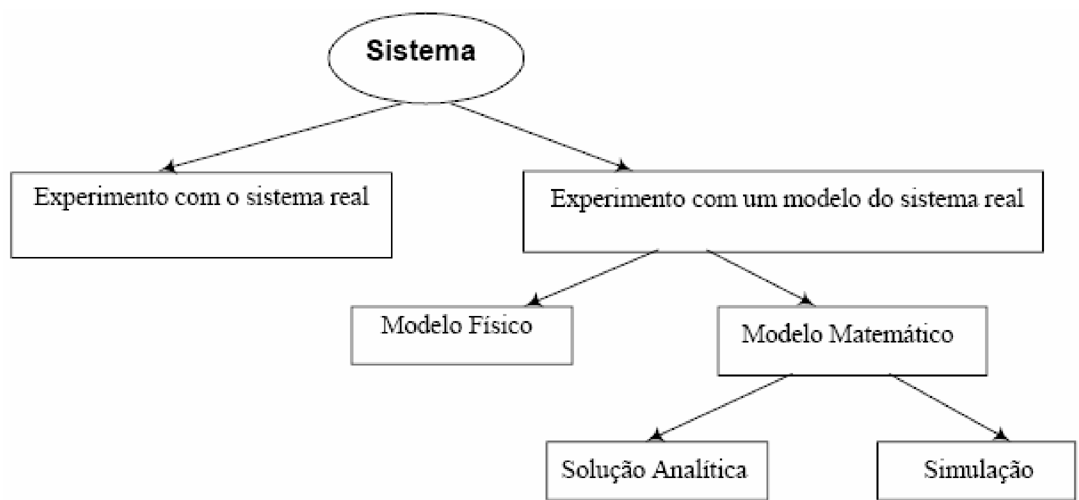

Fig 2. Formas de estudo de um sistema.

Os modelos podem ser ainda classificados como estáticos ou dinâmicos, determinísticos ou estocásticos e discretos ou contínuos.

- Os modelos dinâmicos levam em consideração o tempo. Nos modelos estáticos o tempo não apresenta um papel importante.

- Os modelos determinísticos têm um conjunto conhecido de entradas, os quais resultarão em um único conjunto de saídas, estes não apresentam variáveis aleatórias. Os modelos estocásticos possuem uma ou mais variáveis aleatórias como entrada, que levam a saídas aleatórias. Assim, as saídas da simulação estocástica devem ser tratadas como estimativas estatísticas das características reais de um sistema.

- Os modelos de simulação discreta são aqueles em que as variáveis dependentes (variáveis de estado) variam discretamente em pontos específicos do tempo simulado. Os modelos contínuos são aqueles em que as variáveis dependentes podem variar continuamente ao longo do tempo simulado.

O enfoque deste trabalho limita-se à simulação computacional de um sistemas dinâmicos, discretos e estocástico.

2.4 Conceitos Básicos da Modelagem e Simulação

Há terminologias e conceitos importantes que são comuns na modelagem e simulação de sistemas. A seguir apresenta-se agora um resumo destas terminologias e conceitos relacionados à simulação [5].

Variáveis: são valores globais do sistema, visíveis a qualquer momento e a qualquer ponto do modelo. Por exemplo: o tempo de simulação é uma variável global de um sistema de tráfego.

Variáveis de estado: as variáveis de estado fornecem um conjunto de informações necessárias à compreensão do que está ocorrendo no sistema num determinado momento. Quando a execução de uma simulação é interrompida, é possível continuá-la do ponto que parou se, e somente se, os valores das variáveis de estado forem conhecidos. Além do mais, elas ajudam a determinar se o tipo de modelo é discreto ou contínuo. Um exemplo de variável de estado é a quantidade de carros na estrada em um determinado instante da simulação.

Eventos: toda mudança no estado do sistema é feita pela ocorrência de um evento. Os eventos podem ser programados ou não. Quando ocorrem provocam mudanças nas variáveis ou atributos. Um exemplo de um evento é à entrada de um carro no sistema de tráfego, ou a sua paragem em um semáforo.

Entidades: são elementos do sistema que podem ser distinguidos individualmente dentro do modelo. As entidades podem ser dinâmicas ou estáticas. No caso de dinâmicas, elas levam em consideração o tempo. Um exemplo de entidade dinâmica é o carro. Esta entidade circula pelo sistema sofrendo ações de mover-se pela estrada ou parar nos semáforos. As entidades também podem ser estáticas, servindo a outras entidades, e nesse caso são chamadas de recursos.

Recursos: são entidades estáticas que provêm serviços às entidades dinâmicas. Por exemplo: Os veículos são "criados" e "lançados" no sistema se a estrada estiver criada.

Atributos: são as características próprias de cada entidade. No caso da entidade carro, a placa é um atributo que o identifica, no caso de um semáforo a cor também pode ser vista como um atributo. Os atributos podem atrelar valores importantes a serem transportados ao longo da simulação.

Processos: os processos são ações realizadas sobre a entidade ao longo da simulação, por exemplo, a parada de um carro.

Tempo simulado e tempo de simulação: refere-se ao tempo real, ao relógio de parede. Enquanto o tempo de simulação refere-se ao o tempo necessário à execução de uma "rodada" de simulação no computador, também chamado de tempo lógico.

2.5 Ferramentas de Simulação

2.5.1 Evolução das linguagens de simulação

Os primeiros programas de simulação utilizavam linguagens de propósito geral, como FORTRAN (ver Tabela 1 [11]). Os programas eram grandes e perdia-se muito tempo na elaboração de rotinas e na procura de erros, mesmo em sistemas simplificados. Essas linguagens exigem um forte conhecimento de programação, apesar de, por outro lado, darem maior flexibilidade de programação, permitindo simular-se uma grande diversidade de sistemas.

Tabela 1. Histórico do uso da simulação computacional.

Anos	Ferramentas	Características do estudo de simulação	Exemplos
50 e 60	Linguagens de propósito geral	Aplicadas em grandes corporações; Os grupos de desenvolvimento de modelos com 6 a 12 pessoas. Grandes investimentos em capital; Aplicáveis a qualquer contexto; Exigem conhecimento profundo da linguagem; Exigem muito tempo de desenvolvimento; Não são totalmente reutilizáveis.	FORTRAN
70 e início dos 80	Linguagens de simulação	Utilização em um maior número de corporações; Desenvolvimento e uso de pacotes de linguagens; Linguagens de simulação baseadas em sistemas dinâmicos; Comandos projetados para tratar lógica de filas e demais fenômenos comuns; Mais amigáveis, mas ainda requerem programadores especializados.	SIMSCRIPT, GPSS, GASP IV, DYNAMO, SIMAN, SLAM e etc.
80 e início dos 90	Simuladores de alto nível	Introdução do PC e da animação; Presença de guias e caixas de diálogos; Facilidade de uso; Menos flexíveis que as linguagens de propósito geral e de simulação; Projetados para permitir modelagem rápida; Dispõem de elementos específicos para representar filas, transportadores etc.	Simfactory, Xcell, SIMAN, CINEMA e etc.
Após 90	Pacotes flexíveis de linguagens de simulação	Melhor animação e facilidade de uso; Fácil integração com outras linguagens de programação; Grande uso em serviços; Uso para controlar sistemas reais; Aprimoramento dos simuladores, o que permite modelagem rápida.	Wintness, Extend, Stella, Arena e etc.

A década de 70 foi chamada de “década de ouro” da simulação [17] devido à enorme divulgação desta técnica. Nessa época surgiram linguagens de simulação tais como GPSS, SIMSCRIPT, SLAM e SIMAN. Estas linguagens proporcionaram uma melhor plataforma de programação para simulação, separando os problemas em dois segmentos: modelos e experimentos.

A partir dos anos 80 os computadores permitiram o surgimento de várias ferramentas de simulação, que exploraram as suas potencialidades. Teve-se o surgimento da chamada “simulação visual” [17], com recursos de animação gráfica para os diversos componentes analisados, movendo-os conforme as mudanças de estado do sistema. A partir da década de 90 vários programas com esta característica surgiram: Arena, Stella, Wintness, etc.

À medida que as ferramentas foram evoluindo o tempo de desenvolvimento foi diminuindo. As ferramentas continuam a evoluir, tornando-se cada vez mais adaptáveis, flexíveis e fáceis de usar, além de apresentarem melhores recursos gráficos, de comunicação e interação com o usuário, estatísticos, de animação e etc.

Apesar da simulação apresentar diferentes abordagens para sua implementação neste trabalho trataremos apenas de dois tipos: linguagens de propósito geral e ambientes de simulação. Entre as várias ferramentas disponíveis para cada uma destas abordagens foram escolhidas a linguagem de programação Java e o ambiente de simulação genérico Arena. Suas escolhas devem-se ao fato de serem duas ferramentas bastante conhecidas no meio acadêmico e comercial, e da disponibilidade de referências bibliográficas. Nas seções a seguir, serão detalhadas as características de cada uma, Linguagens de propósito geral e Ambientes de simulação genéricos, respectivamente.

2.5.2 Linguagens de propósito geral

As linguagens de uso geral, tais como: C, C++, FORTRAN e JAVA, permitem implementar os modelos de simulação de forma mais eficiente. Segundo T. Naylor [13] essas linguagens dão suporte a uma maior flexibilidade na construção do modelo. Alguns motivos para o uso das linguagens de propósito geral são [20]:

- Flexibilidade oferecida na descrição matemática do sistema modelado; não há estruturas inerentes limitantes;
- O programador pode selecionar um tipo ou formato de relatório de saída. Somente o tempo e a habilidade de programação limitam os relatórios que podem ser gerados;
- Flexibilidade em termos de tipos de experimentos que podem ser desempenhados no sistema modelado.

De acordo com T. Naylor [13], o maior problema da modelagem em linguagens de propósito geral é a dificuldade de implementação do modelo. Isto se deve ao fato de ficar por conta do modelador todo o controle de seqüência da execução da simulação, esta abordagem dá espaço ao aparecimento de pequenos erros, responsáveis pelo surgimento de efeitos obscuros e difíceis de tratar.

A Fig 3 [4] ilustra como modelos são desenvolvidos em uma linguagem de propósito geral. O modelador pensa no problema em sua linguagem nativa. O resultado é uma formulação

do problema e uma descrição do modelo. O modelador desenvolve um modelo matemático adequado, e em seguida traduz o modelo em um programa de simulação.

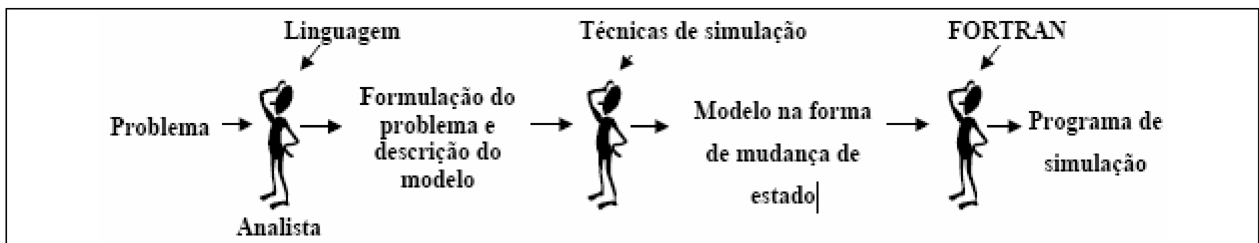

Fig 3. Modelagem com uma linguagem de propósito geral.

As linguagens C, C++ e Java já dispõem de bibliotecas específicas para simulação, que são respectivamente SIMLIB para C e C++ , e JAVASIM, entre outras, para Java. A opção de não usar bibliotecas de simulação, no nosso estudo, porque elas possuem embutidos blocos que ajudam na implementação do modelo. Um dos objetivos do trabalho é deixar o modelador responsável por todo controle de seqüência da simulação.

A linguagem Java foi escolhida para programar o estudo de caso deste trabalho, por ser uma das linguagens de propósito geral mais utilizada atualmente, e também porque existe uma grande quantidade de material disponível em livros e na internet.

2.5.3 Ambientes de Simulação

Ambientes de simulação geralmente são mais amigáveis e dedicados apenas à simulação, além de disponibilizarem recursos adicionais para visualização e animação da simulação, assim como para o tratamento de dados de entrada e saída. Na verdade, os ambientes são bibliotecas compostas por macros de linguagens de simulação.

Os ambientes de simulação podem ser genéricos ou específicos. Os ambiente genéricos são projetados para a modelagem de sistemas de vários tipos, como é o caso do Arena, já os específicos são projetados para uma determinada área de interesse (exemplo: o NS para modelagem de redes), dessa forma apresentam características específicas implementadas mais poderosas, porém não podem representar qualquer modelo ou área de interesse.

A maioria dos ambientes de simulação genérico vê o problema a ser modelado como um conjunto de estações que prestam serviços às entidades. Segundo T. Naylon [13], esses ambiente são menos gerais e flexíveis do que as linguagem de propósito geral, uma vez que estão restritos às funcionalidades implementadas para simulação, foco a sua praticidade de uso. Em contrapartida são mais flexíveis que ambientes de simulação de propósito específicos.

A Fig 4 mostra os passos de um modelador para desenvolver o modelo utilizando um ambiente de simulação genérico, que são diferentes daqueles utilizados numa linguagem de propósito geral. O analista pensa no problema em função dos blocos de controle que a ferramenta dispõe, eliminando assim a etapa de descrição do problema. Em seguida o computador é utilizado para o processo de tradução produzindo o programa de simulação.

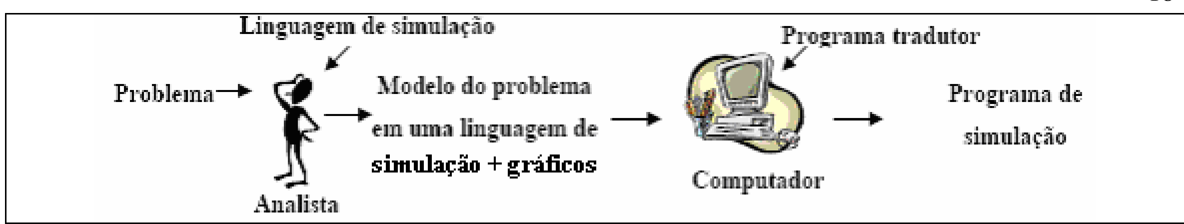

Fig 4. Modelagem com um ambiente de simulação.

As vantagens de se utilizar os ambientes de simulação genérico estão diretamente ligadas às desvantagens de se utilizar às linguagens de programação, e vice-versa; quando utilizamos o ambiente de simulação ganhamos na praticidade, pois a maiorias das tarefas podem ser feitas utilizando apenas o *mouse* (reduzindo a quantidade de programação). Uma grande quantidade de módulos de controlem já estão implementados, dentre outras vantagens. Todavia, estes ambientes perdem na representação dos modelos, em virtude destas estarem limitadas a uma visão de mundo de quem projetou o ambiente.

O ambiente de simulação genérico adotado para este trabalho foi o Arena 9 *Academic*, disponível na internet para uso e livre de pagamentos, todavia esta versão possui uma limitação no número de entidades que podem ser criadas simultaneamente.

2.5.4 O ambiente de Desenvolvimento Arena

Em 1993, a empresa americana *Systems Modeling* lançou o Arena, substituindo dois de seus produtos: o SIMAN e o CINEMA. O primeiro era responsável pela simulação, enquanto o CINEMA fazia a animação gráfica do modelo. O Arena unificou os dois programas, aperfeiçoando-os. O Arena possui uma arquitetura modular que permite a construção de modelos através de blocos. Estes blocos representam partes de um todo que podem ser interconectados de forma a modelar o sistema real. Com uma interface bastante amigável e interativa, pode-se facilmente e rapidamente fazer a modelagem. Existem duas partes básicas do Arena: a lógica e a animação. A primeira é constituída dos blocos de simulação que montam todo o sistema a ser modelado. A segunda é à parte da animação gráfica. Dados os blocos montados da parte lógica, são anexados a estes desenhos e símbolos para a visualização do sistema real durante a evolução do tempo na simulação. A versão 9 do Arena já vem com recursos voltados à animação em três dimensões, mostrando que este ambiente está em constante evolução.

O Arena possui quatro ferramentas básicas que são bastante úteis:

- Arena Viewer: visualizador da simulação. Permite que o modelo executado já construído e previamente preparado seja executado no computador sem instalação de todo o programa do Arena.

- Input Analyser: analisador de dados de entrada. Permite análise de dados do sistema e a escolha de uma distribuição estatística que melhor se adeque ao problema.

- OutPut: Analyser: analisador de dados de saída. Permite a visualização dos dados coletados durante a simulação.

- OptQuest: emprega técnicas de otimização no sistema modelado na busca de soluções ótimas e viáveis, considerando as inúmeras combinações de parâmetros.

O Arena fornece ao usuário uma visão geral do sistema, além de facilitar a modelagem e economizar tempo de projeto. É mais simples alterar o modelo depois de escrito e mais fácil de resolver possíveis problemas. Possuem recursos avançados para modelar vários tipos de sistemas.

A Tabela 2 [18] compara as vantagens e desvantagens dessas duas ferramentas utilizadas para simulação.

Tabela 2. Comparação das vantagens e desvantagens das linguagens.

Linguagens de Propósito Geral	
Vantagens	**Desvantagens**
Pequeno número de restrições impostas ao formato de saída.	Tempo de programação mais longo.
Freqüentemente há conhecimento prévio da linguagem a ser utilizada.	A solução de problemas na simulação pode não ser simples.
Ambiente de Simulação	
Vantagens	**Desvantagens**
Requerem menor tempo de programação	As saídas são definidas pelas características inerentes ao programa utilizado.
Fornecem técnicas de verificação de erros superiores àquelas disponíveis nas linguagens de propósito gerais.	Flexibilidade reduzida e maior tempo de execução.
Fornecem um veículo breve e direto para expressar conceitos conscientes ao estudo da simulação.	
Habilidade de construir e fornecer aos usuários sub-rotinas requeridas como parte de qualquer rotina de simulação.	
Facilitam a coleta e exibição dos dados produzidos.	
Controlam o gerenciamento e alocação da capacidade de armazenamento do computador durante a execução da simulação.	

2.6 Procedimento para um estudo de Modelagem e Simulação

Um estudo de simulação baseia-se numa seqüência de passos que podem ser chamadas de metodologia de simulação a qual envolve quatro fases, chamadas neste trabalho de: i) Descobrimento e Orientação, ii) Construção do modelo e coleta de dados, iii) Execução do modelo, iv) Implementação. Esta metodologia é clássica, ou seja, encontra-se em quase todos os livros e trabalhos relacionados com modelagem e simulação. Este trabalho baseou-se nos textos de [5] [16] [17] para apresentar a Fig 5 e descrever os passos de como proceder para solucionar um problema através da modelagem e simulação.

1° Fase – Descobrimento e Orientação

Formulação do problema a ser simulado: todo estudo deve começar pela definição do problema. Se as definições são realizadas pelo usuário que está com o problema, o projetista deve se assegurar de que o problema foi efetivamente entendido. Se as definições do problema são desenvolvidas pelo projetista, é importante que o usuário também esteja de acordo com a formulação. Os objetivos devem estar de modo bem claro nesta etapa. Questões do tipo: Por que estudar o problema? Que respostas esperam-se alcançar? Que limites e restrições existem no sistema? Ajudam a formular o problema.

Plano de Projeto e Conjunto de Objetivos: o projeto pode incluir os planos de estudo em termos do número de pessoas envolvidas, do custo do estudo e do número de dias necessários para concluir a fase de desenvolvimento com os resultados antecipados de cada estágio no desenvolvimento. Nesse passo, deve-se ter um cronograma de atividades. Descreve-se também o que deve ser representado no modelo e deve-se determinar o método que será utilizado para as medidas de desempenho, como por exemplo, solução analítica ou por simulação.

2° Fase – Construção do modelo e Coleta de Dados

Concepção do modelo: a tarefa de modelagem envolve certa habilidade para abstrair as características essenciais de um problema, para selecionar e modificar as suposições que caracterizam o sistema, e para destacar os resultados de interesse. Deve-se começar com um modelo simples, e então, a partir deste, chegar a modelos mais complexos. A complexidade do modelo não deve ser maior que aquela requerida para alcançar os objetivos do estudo. A violação deste princípio aumenta os custos de construção do modelo e de execução do modelo. Não é necessário ter uma correspondência perfeita entre o modelo e o sistema real. Apenas a essência do sistema real é necessária no modelo. É aconselhável envolver o usuário na concepção do modelo. Isto aumenta a qualidade do modelo resultante e a confiança deste usuário na aplicação do modelo.

Coleta de Dados: com a construção do modelo completa, é possível identificar os parâmetros (dados) que serão necessários. Esses parâmetros podem ser classificados como parâmetros de carga (como por exemplo, tempo entre chegadas de novos clientes ao sistema, tempo de execução, tipos e tamanhos de registros) e parâmetros do sistema (tempo para operações de acesso à memória, por exemplo). Caso o sistema real ainda não exista, pode-se utilizar os parâmetros de um sistema existente, com características semelhantes.

Tradução do modelo: os sistemas do mundo real podem resultar em modelos que envolvem uma grande quantidade e variedade de informações, assim os modelos precisam ser traduzidos para um formato adequado para serem tratados num computador. O modelador precisa decidir qual ferramenta irá usar para implementar o modelo. Na sessão 2.5, podem ser observados vários conceitos de ferramentas para abordar um modelo, principalmente ambiente de simulação genérico e linguagem de propósito de geral. Contudo o modelador deve usar àquela que melhor se adaptar ao problema estudado, em virtude de todas apresentarem vantagens e desvantagens.

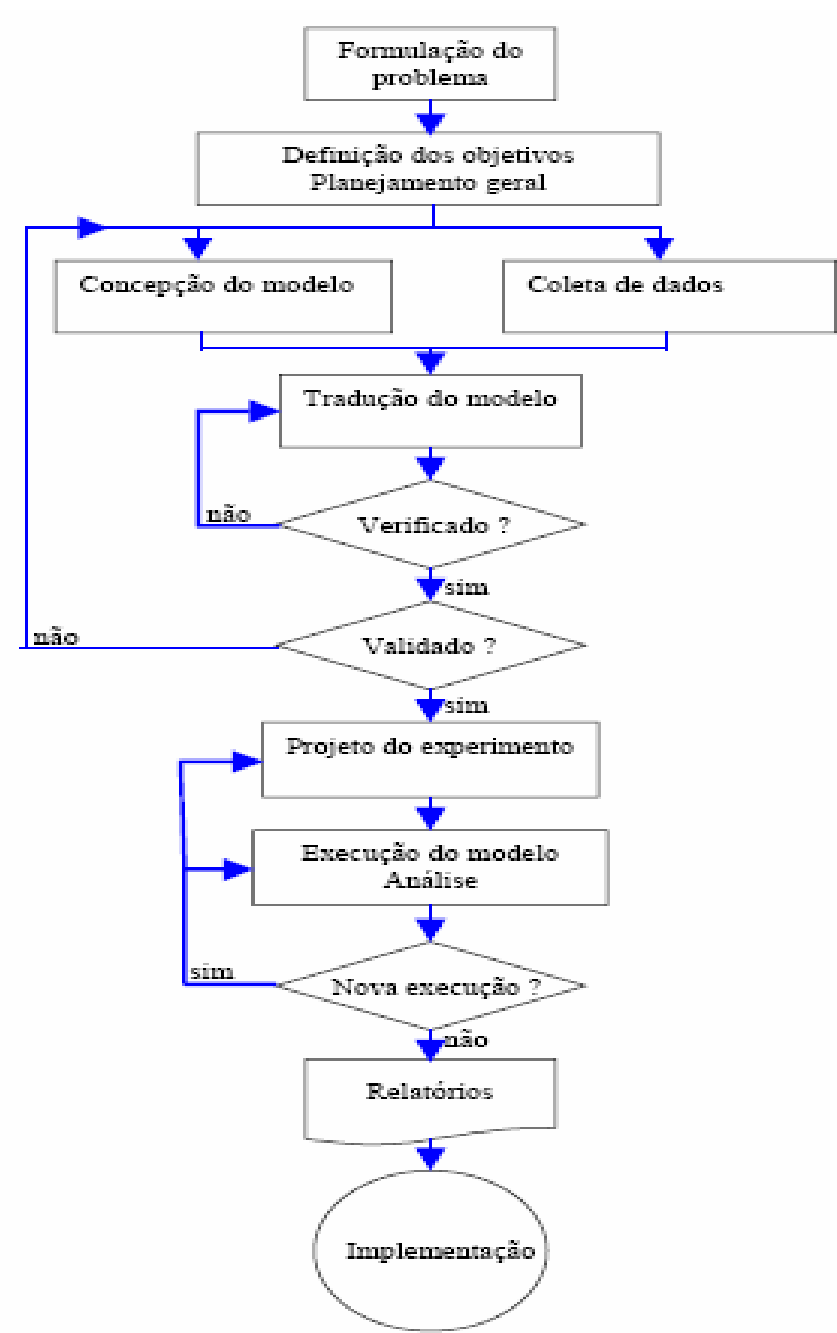

Fig 5. Passos para modelar um problema utilizando modelagem e simulação.

Verificação: verifica se o programa de computador executa o modelo da forma esperada. Quanto mais complexo for o modelo maior os riscos de erro na tradução para o computador. A verificação faz parte do processo de edição e programação do mesmo.

Validação: é a confirmação de que um modelo é uma representação adequada do sistema real. Validação é um processo interativo de comparar dados do modelo com o comportamento do sistema real, usando as diferenças entre os dois para melhorar o modelo. Este processo é repetido até que o modelo seja julgado aceitável.

3° Fase – Execução do Modelo

Projeto Experimental: envolve determinar as alternativas que serão simuladas. Para cada projeto do sistema que é simulado, é necessário tomar decisões para relacionar o tempo necessário para a iniciação, o tempo necessário para a execução da simulação e qual o método para análise de saída que será utilizado.

Execução do modelo e análise: a execução do modelo e sua subseqüente análise são realizadas e utilizadas para estimar medidas de desempenho para o sistema que está sendo simulado.

Análise da execução: baseado na análise das execuções realizadas determina-se a necessidade experimentos adicionais e se novas especificações devem ser consideradas.

4° Fase - Implementação

Documentação: quando os resultados são considerados aceitáveis, faz-se a documentação ajudando assim o trabalho de manutenção e possíveis modificações futuras no modelo, prolongando a vida útil do modelo.

Implementação: o sucesso da fase de implementação depende de como foram conduzidos os passos anteriores. Se o usuário esteve envolvido durante o processo de edição do modelo e entende a natureza do modelo e suas saídas, este poderá contribuir efetivamente para a fase de implementação.

2.7 Simulação Distribuída

Com o aumento do uso de simulação de sistemas por computador, sistemas cada vez mais complexos passaram a ser simulados, tornando assim a simulação seqüencial custosa, em virtude do tempo de execução tornar-se praticamente inviável. Para minimizar esse tempo de simulação pode-se dividir os processos e executa-los em paralelo. A simulação distribuída baseia-se em executar um programa de simulação em um ambiente de computação contendo múltiplos processadores. Tal ambiente de computação pode ser: um conjunto de processadores dentro de uma mesma máquina ou máquinas geograficamente distribuídas, interligadas por uma rede de comunicação [6]. A simulação distribuída vem sendo aplicadas em várias áreas como: controle de tráfego aéreo, treinamentos militares promovidos através da simulação, simuladores de vôos, sistemas de transportes, simuladores de jogos, etc. [14] [21]

Além de reduzir o tempo de processamento, a simulação distribuída apresenta outras vantagens [6], entre elas:

- Máquinas de diferentes arquiteturas podem integrar seus simuladores.

- Distribuição geográfica: se ocorrer problemas em algum processador e o mesmo parar de funcionar, outros processadores podem executar as tarefas que aquele processador estava executando.

Em processamento paralelo pode existir erros de inconsistência de dados, ou seja, o tempo em que um evento ocorre em um componente pode interferir no tempo de ocorrência de um evento em outro componente. Portanto surgiu a necessidade de se criar protocolos que garantam a coerência dos dados e a execução correta da simulação. Esses protocolos são chamados protocolos de sincronização e são classificados em dois tipos síncronos e assíncronos. Os protocolos síncronos possuem um mecanismo geral para o controle da simulação, através do qual os processos dividem o mesmo relógio geral. Nos protocolos assíncronos cada processo possui um relógio único e sincroniza com os processos com os quais se comunica. Esses protocolos dividem-se em conservativos, otimistas e os mistos [6], conforme ilustrado na Fig 6.

- O protocolo tempo conservativo: os tempos de ocorrências dos eventos são verificados para que haja a certeza de que nenhum evento seja executado fora da ordem cronológica. Desse modo o protocolo garante que os eventos da simulação são executados na ordem imposta pelo sistema que está sendo simulado.

- O protocolo otimista os eventos são executados sem que haja nenhuma verificação quanto à ordem. Quando ocorre um erro de causa e efeito, um mecanismo de *rollback é* utilizado para recuperar o estado da simulação para um estado consistente. A simulação continua a execução dos eventos partindo do estado recuperado da simulação.

- Além desses dois tipos de protocolos, temos a combinação dos mesmos, nos quais o sistema ora executa de maneira otimista, ora executa em modo conservativo. Esses protocolos são conhecidos como protocolos mistos.

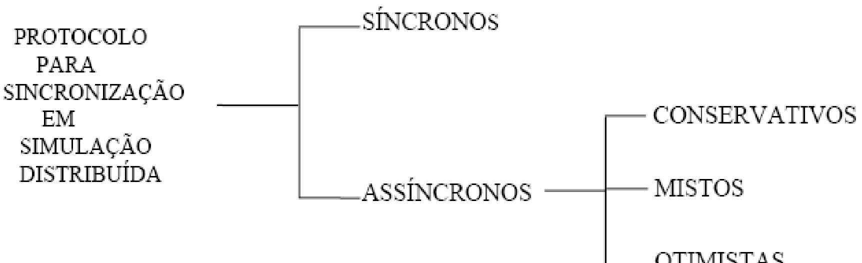

Fig 6. Classificação dos protocolos de simulação distribuída.

Assim como a simulação seqüencial, a simulação distribuída possui linguagens que facilitam o desenvolvimento de projetos, fornecendo um conjunto completo de primitivas para os projetistas modelarem seus sistemas. Escondendo as características dos protocolos de sincronização. Algumas linguagens de simulação distribuída são [12]: Apostle (*A Parallel Object-oriented SimulaTion Language*); Maisie: Moose (*Maisie-based Object-Oriented Simulation Environment*); Parsec (*Parallel Simulation Environment for Complex Systems*); ModSim; Yaddes, etc.

Além das linguagens de simulação distribuída também existem bibliotecas, nas linguagens de propósito geral, dedicadas à simulação distribuída. As quais tem o propósito de reduzir o tempo de desenvolvimento, entre elas [2] [12]: *GTW* e *TWOS* bibliotecas da linguagem C; *PSK*, *SPaDES*, *SPEEDES* e *WARPED* orientadas a objetos utilizando C++; JAVASIM pacote da linguagem Java e entre outras.

Existem também ambientes de simulação distribuída automático, os quais disponibilizam uma série de recursos para a construção da simulação, fazendo também uso de programação visual. Os modelos são feitos na tela e os resultados exibidos por gráficos e animações. Entre eles pode-se citar: UCLA *Simulation Environment*; Akaroa-2; ASDA, etc.

2.8 Simulação na *web*

Existe um grande interesse de usar a *web* como uma nova plataforma de aplicações [10]. Devido à proliferação da internet a simulação tem sido forçada a migrar para *web*. A primeira vez que o assunto foi abordado foi em 1996, na *Winter Simulation Conference*, provocando grande interesse da comunidade científica, e desde então seu interesse vem aumentando desde então. Em 1998, foi definido Java como a linguagem fundamental e a arquitetura CORBA para modelagem e simulação baseada em *web*. CORBA é uma especificação responsável por gerenciar a comunicação entre os objetos distribuídos.

Existem diversos tipos de aplicações que podem utilizar as vantagens de computar pela internet, os mais comuns são:

- Aplicações que tratam com quantidades de dados enormes, como por exemplo, modelos meteorológicos.

- Aplicações que permitem usuários de muitos locais diferentes colaborar no desenvolvimento de um projeto.

- Aplicações na instrução e treinamento de estudantes em programas de ensino à distância.

Foram desenvolvidos vários ambientes de simulação discreta baseada em Java. Ferramentas que suportam descrição de modelos textuais e gráficos. Seguem alguns exemplos para construção de modelos: simjava, DEVSJAVA, JSIM, JavaSim, JavaGPSS, Silk e WSE [10].

2.9 Arquitetura de Alto Nível para Simulação

O HLA foi desenvolvido pelo US DoD (U.S. *Department of Defense*) num processo de esforço conjunto envolvendo o governo, o ambiente acadêmico e a indústria. Em 1995, três equipes de indústria desenvolveram conceitos para a definição de uma arquitetura alto nível. Os resultados foram combinados com considerações adicionais de outros projetos de modelagem e simulação. No dia 31 de março de 1995, esta definição foi apresentada à DMSO (*Defense Modeling and Simulation Office*) e ao AMG (*Architecture Management Group)*. O resultado constituiu a base da definição do HLA, completada em agosto de 1996, e foi aprovada como técnica de arquitetura padrão para todas as simulações do US DoD, em 10 de Setembro de 1996. O AMG continua evoluindo o HLA baseado nas experiências com seu uso. Esta especificação formou a base para um desenho *draft* padrão do IEEE (*Institute of Electrical and Eletronics Engineers*) para arquitetura de interoperabilidade de simulação [3].

O HLA é o padrão prescrito para interoperabilidade de simulação militar dentro do EUA, na OTAN como também em muitos países, por exemplo, a Suécia. O HLA pode ser usado por qualquer sistema de simulação. Há um interesse crescente de áreas não militares como, por exemplo, a área de transportes.

HLA é um padrão que pode conectar vários sistemas de simulação computacionais de forma que estes possam executar em paralelo e trocar informações. Em vez de construir um sistema de simulação monolítico grande, o HLA tem como objetivo permitir combinar sistemas de simulação menores. HLA permite usar novos sistemas existentes para novos propósitos, permitindo também misturar diferentes linguagens de programação e sistemas operacionais.

O HLA baseia-se nas seguintes premissas e suposições: (1) Uma simulação não pode satisfazer todos os usos e usuários, (2) Desenvolvedores possuem conhecimentos diferentes da simulação, e nenhum desenvolvedor é especialista em todos os detalhes da simulação, (3) Ninguém pode antecipar todos os usos de simulação e todos os modos podem ser completamente combinados, (4) Tecnologias futuras e ferramentas devem ser incorporadas. Uma simulação ou um conjunto de simulações desenvolvidas para um propósito específico poderia ajudar outra simulação. Visando isso, o HLA fornece uma interface de comunicação padrão para os modelos de simulação e seus componentes e uma série de serviços. A intenção do HLA é fornecer uma estrutura a qual suportará reuso das capacidades disponíveis em diferentes simulações, reduzindo custos e tempo para o desenvolvimento do novo ambiente.

Através do HLA deve-se ser possível (1) Decompor um grande problema a ser simulado em partes menores, pois assim a definição, a construção e a verificação são facilitadas. (2) Compor diversas simulações com o objetivo de se construir um sistema maior, e mais complexo, a partir de simulações menores. (3) Separar as funções genéricas das específicas, possibilitando assim reutilizar as funções genéricas nas próximas simulações. (4) A partir da interface entre simulações e as funções genéricas, separar as simulações das trocas de tecnologias [9]. Para alcançar os objetivos acima o HLA necessita usa componentes modulares com funcionalidade e interface bem definida, mantendo a liberdade necessária para simulações individuais sem comprometer a interoperabilidade pedida entre as simulações [3].

Ainda com o propósito de manter sua generalidade, o HLA não é implementado em nenhuma linguagem de programação específica, porque se acredita que com o avanço tecnológico novas linguagens irão surgir, e novas e diferentes implementações serão possíveis.

Eis alguns conceitos importantes sobre HLA:

- Federação: Conjunto de simulações com o mesmo FOM (*Federation Object Model*);

- Federado: Membro da federação que representa um ponto de conexão com a infra-estrutura. Pode ser um modelo simples (automóvel) como também agregar uma simulação maior (controle de tráfego aéreo);

- SOM (*Simulation Object Model*): Descreve o que cada federado pode produzir ou consumir;

- FOM (*Federation Object Model*): define o que existe de comum nos federados (para ser usado na simulação).

O conceito de federados é que permite utilizar componentes modulares, possibilitando organizar a definição de interface e funcionalidade. Um federado pode ser uma simulação, coletores/visualizadores e objetos com interface com o mundo real. Portanto é necessário que os

federados possuam características que façam a comunicação com outras simulações através de trocas de mensagens, a qual utiliza a RTI (*Runtime Infrastructure*).

***Runtime Infrastructure* (RTI)**

O RTI foi desenvolvido como parte do HLA. Ele permite aos federados que estão participando de uma simulação conectarem-se um ao outro e trocar informações. O RTI pode informar quais objetos estão conectados e quais os valores de seus atributos.

O RTI pode sincronizar o tempo dentro da federação e provê funções mais avançadas como transferir a responsabilidade de atualizar um atributo dentro de um federado. Fica a cargo dos programadores implementar os serviços desejados.

Capítulo 3

Estudo de Caso

O estudo de caso realizado neste trabalho será um sistema de tráfego de automóveis fictício. A área deste estudo de caso é motivada pela riqueza de informações que podem ser exploradas, abordando a maioria dos conceitos estudados ao longo do texto. Esta fase do trabalho compreende a elaboração do estudo de caso, onde foram realizados os passos: descobrimento e orientação, construção do modelo e coleta de dados, e execução do modelo; dos procedimentos de um estudo de simulação exposto na sessão 2.6.

Para isto foi utilizados um computador com processador Athlon 2000 com 256 MB de memória RAM. Foi usado a versão *Academic* do Arena 9.0, a qual possui algumas limitações quanto à quantidade de entidades no sistema, que não poderá ultrapassar 150 entidades simultâneas. Também foi utilizada a versão J2SDK1.4.1_02 do Java e a versão 3.0 do Eclipse. Foram despendidos cerca de três meses com o aprendizado do Arena, sendo construídos modelos pequenos, que foram constituindo subsistemas do estudo de caso.

3.1 Descobrimento e Orientação

Nesta fase definimos o problema real a ser modelado e estudado. A partir desta descrição será definido o modelo conceitual, que representará o sistema original mantendo as características essências para sua concepção.

3.1.1 Formulação do Problema

O problema a ser estudado refere-se a um sistema de tráfego automotivo que está representado na Fig 7. O sistema é composto de três estradas, chamadas de Estrada 1, Estrada 2 e Estrada 3. As estradas possuem comprimento e largura que limitam suas dimensões. A Estrada 2 e a Estrada 3 cortam a Estrada 1, cada uma em um ponto diferente, formando dois cruzamentos. Cada cruzamento possui um semáforo responsável por controlar o tráfego de veículos naquela região.

Os semáforos são compostos de dois sub-semáforos, onde cada sub-semáforo possui três luzes: verde, amarela e vermelha. Cada cor possui um tempo de ativação distinto. As cores de um mesmo conjunto são sincronizadas entre si (seguindo a seqüência: verde → amarela → vermelha e volta para o verde retomando a seqüência) e os sub-semáforos (contidos num mesmo

semáforo) também são sincronizados. Por exemplo, quando um sub-semáforo de um semáforo encontra-se com a cor verde ativa, o outro sub-semáforo está com a cor vermelha acionada. A Fig 8 mostra uma seqüência de mudança de cores entre os dois sub-semáforos de um mesmo semáforo.

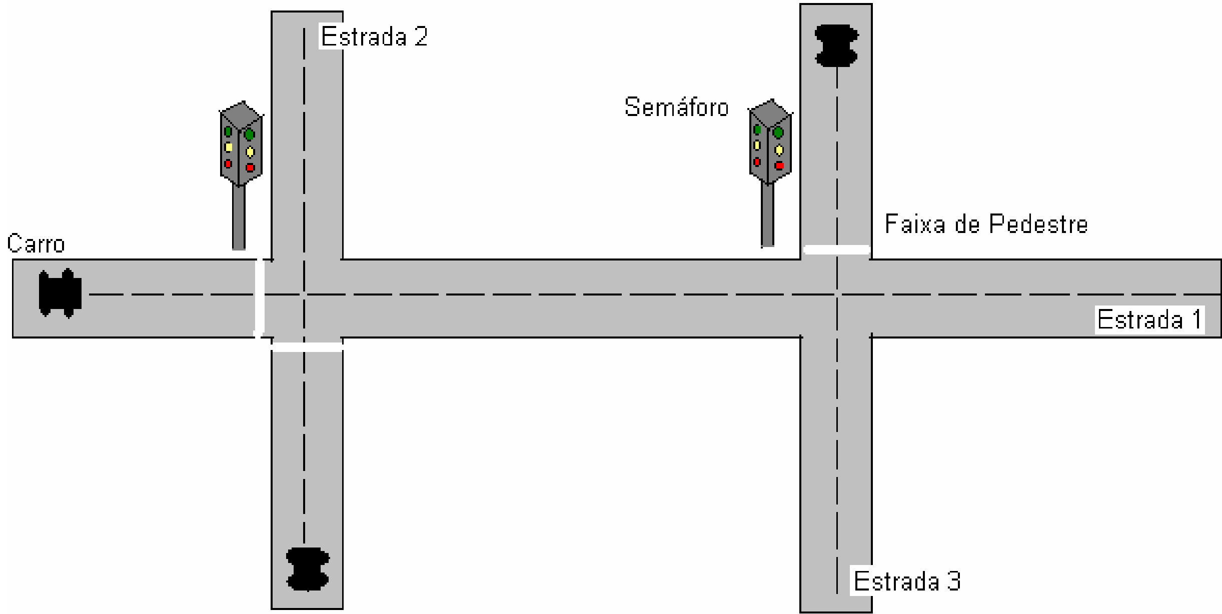

Fig 7. Sistema simulado.

Os carros chegam através das três estradas com uma dada determinada velocidade inicial. O ritmo de chegada dos carros depende do horário, ou seja, há horários em que o tráfego de carros é intenso, horários em que o tráfego é considerado normal, e ainda horários em que quase não há tráfego de carros. Chamamos estes horários respectivamente de pique, normal e madrugada. Os carros seguem sempre em frente em uma mesma estrada, portanto os carros não podem dobrar à esquerda nem à direita. Os carros mais rápidos ultrapassam os carros mais lentos. Os carros (na perspectiva do motorista) possuem um campo visual, ou seja, a distância máxima a partir da qual podem "ver" um objeto à sua frente. Quando um sub-semáforo surge no campo visual de um carro, este verifica que cor está ativa no momento, se for a cor verde ou amarela o motorista segue seu percurso normalmente, sem alterar a sua velocidade, se for a cor vermelha o motorista pára o seu veículo antes da faixa de pedestre; os outros carros que forem chegando, em seguida, também vão parar antes da faixa de pedestre, formando uma fila de carros. Quando o semáforo "abre", muda para a cor verde, os carros retomam o movimento.

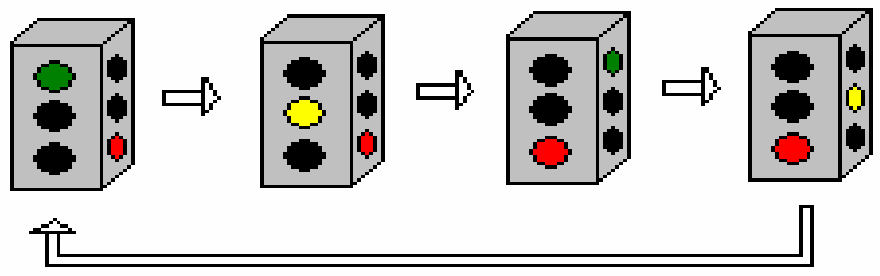

Fig 8. Passos de mudança de cor de um semáforo.

O estudo tem como objetivo avaliar o sistema analisando a melhor forma de programação de seus semáforos, na tentativa de maximizar a sua utilização, ou seja, tornar o tempo de ativação de cada cor o tempo necessário (nem mais nem menos). Com isto o tempo de permanência dos carros nas estradas diminui, aumentando a sua quantidade dos mesmos que circulam pelo sistema.

3.2 Construção do Modelo e Coleta de Dados

Nesta fase, após a descrição do problema real podemos nos concentrar nas características mais importantes para a simulação, descartando níveis de complexidade elevados e níveis de complexidades muito simples. Não é necessário ter uma correspondência perfeita entre o modelo e o sistema real. Apenas a essência do sistema real é necessária no modelo.

Esta fase também contempla a análise de dados de entrada. Quando o sistema real existe deve-se coletar os dados do mesmo, quando não existe dados fictícios devem ser usados para analisar o sistema.

3.2.1 Concepção do Modelo de Tráfego

Na modelagem do problema apresentado, seção 3.1.1, tem a preocupação de garantir níveis apropriados de complexidade do modelo, evitando modelos complicados que apenas acrescentam complexidades computacionais e modelos muito simples que não representam adequadamente o sistema.

Foram identificadas e consideradas as seguintes entidades para a concepção deste modelo:

- **Estrada**: é um recurso (entidade estática) que possui um atributo comprimento.

- **Semáforo**: também é um recurso que possui dois sub-semáforos. Cada sub-semáforo é um conjunto formado pelas luzes (verde, amarela e vermelha). Cada uma destas possui um tempo de ativação próprio. A sincronização ocorre dentro de um mesmo sub-semáforo e entre sub-semáforos de um mesmo semáforo. Por exemplo, num mesmo sub-semáforo as cores seguem a seqüência verde → amarelo → vermelho e volta para o verde, cada cor no seu tempo. A sincronização entre conjuntos de um mesmo semáforo segue os passos de mudança exibidos na Fig 8.

- **Carro**: é uma entidade dinâmica. Os carros entram na estrada com uma dada velocidade inicial. Eles possuem ritmos de chegada dependente do horário. Os horários modelados são pique, normal e madrugada. Os carros também possuem campo visual. Quando o semáforo está "fechado" (cor vermelha ativada) os carros param antes da faixa de pedestre e formam uma fila de carros esperando o semáforo abrir novamente.

Foram identificados e considerados os seguintes processos:

- Mudança de cor do sinal.

- Carro em andamento.

- Carro parado no sinal.
- Mudança de velocidade.

Foram identificadas e consideradas os seguintes eventos:

- Carro entra em uma determinada estrada.
- Carro sai de uma determinada estrada.
- Carro pára movimento.
- Carro inicia andamento.
- Carro identifica algo no seu campo visual.
- Semáforo muda de cor.

Como o estudo tem como objetivo de avaliar a melhor forma de programação de seus semáforos. Para isso foram consideradas as seguintes variáveis de resposta:

- quantidade de carros que passaram pelo sistema.
- Tempo médio que os veículos passaram na fila de cada semáforo, em cada estrada.
- O tempo médio que os veículos passaram no sistema.

Na modelagem deste estudo de caso foram feitas algumas abstrações e restrições na tentativa de torna o modelo viável de ser implementado, valorizando as características essenciais para a implementação nas duas abordagens. Eis algumas limitações:

- Foi desconsiderado a largura das estradas, com isso os carros mais rápidos ultrapassam os carros mais lentos, sem a preocupação com o impacto entre os carros.
- Os carros não dobram à direita nem à esquerda, deslocam-se apenas para frente numa mesma estrada.
- Os carros possuem velocidade constante, ou seja, os carros não têm aceleração nem desaceleração.
- Os motoristas e carros são considerados como sendo uma única entidade, o Carro.

3.2.2 Coleta de Dados

Os dados de entrada são fictícios, não tendo existido coleta de dados real, através do conhecimento do modelador. Foram construídas três tabelas representando os horários de pique, normal e madrugada das Estradas 1, 2 e 3. As Tabelas 3, 4 e 5 mostram respectivamente os ritmos de chegadas dos veículos nas três estradas nos dias úteis da semana (segunda-feira a sexta-feira).

Tabela 3. Intervalo de chegada da Estrada 1.

Horário				Intervalo Entre chegada (Minuto)	Ritmo de Chegada (Quant. Veic.)
De (Horas)	Até (Horas)	Duração (Horas)	Estado		
6:00	8:00	2	Pique	1	4-5
8:00	12:00	4	Normal	2	2-3
12:00	14:00	2	Pique	1	3-5
14:00	18:00	4	Normal	2	1-2
18:00	20:00	2	Pique	1	4-5
20:00	00:00	4	Normal	2	2-4
00:00	6:00	6	Madrugada	60	1-2

Tabela 4. Intervalo de chegada da Estrada 2.

Horário				Intervalo Entre chegada (Minuto)	Ritmo de Chegada (Quant. Veic.)
De (Horas)	Até (Horas)	Duração (Horas)	Estado		
6:00	8:00	2	Pique	1	3-5
8:00	12:00	4	Normal	2	2-3
12:00	14:00	2	Pique	1	4-5
14:00	18:00	4	Normal	2	1-3
18:00	20:00	2	Pique	1	4-5
20:00	00:00	4	Normal	2	2-3
00:00	6:00	6	Madrugada	60	0-2

Tabela 5. Intervalo de chegada da Estrada 3.

Horário				Intervalo Entre chegada (Minuto)	Ritmo de Chegada (Quant. Veic.)
De (Horas)	Até (Horas)	Duração (Horas)	Estado		
6:00	7:00	1	Pique	1	4-5
7:00	12:00	5	Normal	2	2-3
12:00	14:00	2	Pique	2	5-6
14:00	19:00	5	Normal	2	2-3
19:00	00:00	5	Pique	2	4-6
00:00	6:00	6	Madrugada	60	1-2

3.2.3 Tradução do Modelo

Nesta fase o modelador escolhe uma ferramenta para traduzir o modelo. Como faz parte deste trabalho fazer a implementação do modelo utilizando duas abordagens diferentes, serão descritas duas traduções: uma de um modelo construído usando-se o ambiente Arena, e outra de um outro modelo construído usando Java. Primeiramente veremos os passos de desenvolvimento no Arena, em seguida os passos em Java.

3.2.3.1 Tradução do modelo no Arena

No processo de tradução do estudo de caso, identificamos as entidades, recursos, processos e eventos do sistema. A seguir apresentaremos a modelagem do carro, sinal, estrada, e o controlador da simulação.

a) Modelagem do Carro

Como visto no modelo conceitual o carro é uma entidade dinâmica, devido a deslocar-se pelo sistema e é "lançado" nas três estrada. O Arena dispõe de um módulo específico para criar as entidades de entrada do sistema. Tal módulo chama-se *Create* (Criador) conforme pode ser visto na Fig 9. Para este módulo deve ser fornecido o tipo de entidades de criação, o intervalo entre chegada, quantas entidades por chegada, o máximo de entidades por chegada. Neste estudo utilizou-se a distribuição de Poisson. A qual é adequada quando se está modelando as fontes de chagada de um sistema. São numerosos o emprego desta distribuição [5]: requisições feitas a um servidor, chegada de entidades a um sistema entre outras.

Fig 9. Módulo *Create* do Arena.

Em nosso exemplo a chegada das entidades depende do horário. Como o ritmo de chegadas do exemplo é padrão, podemos utilizar o recurso chegada conforme programação *schedule* já existente no módulo *Create* do Arena para modelar o intervalo entre chegadas das entidades para o nosso problema. A Fig 10 exibe um trecho do gráfico de chegada gerado a partir da configuração da opção *Schedule*, de acordo com Tabela 3. No Anexo I pode-se ver os outros gráficos de intervalo de chegada das tabelas 4 e 5.

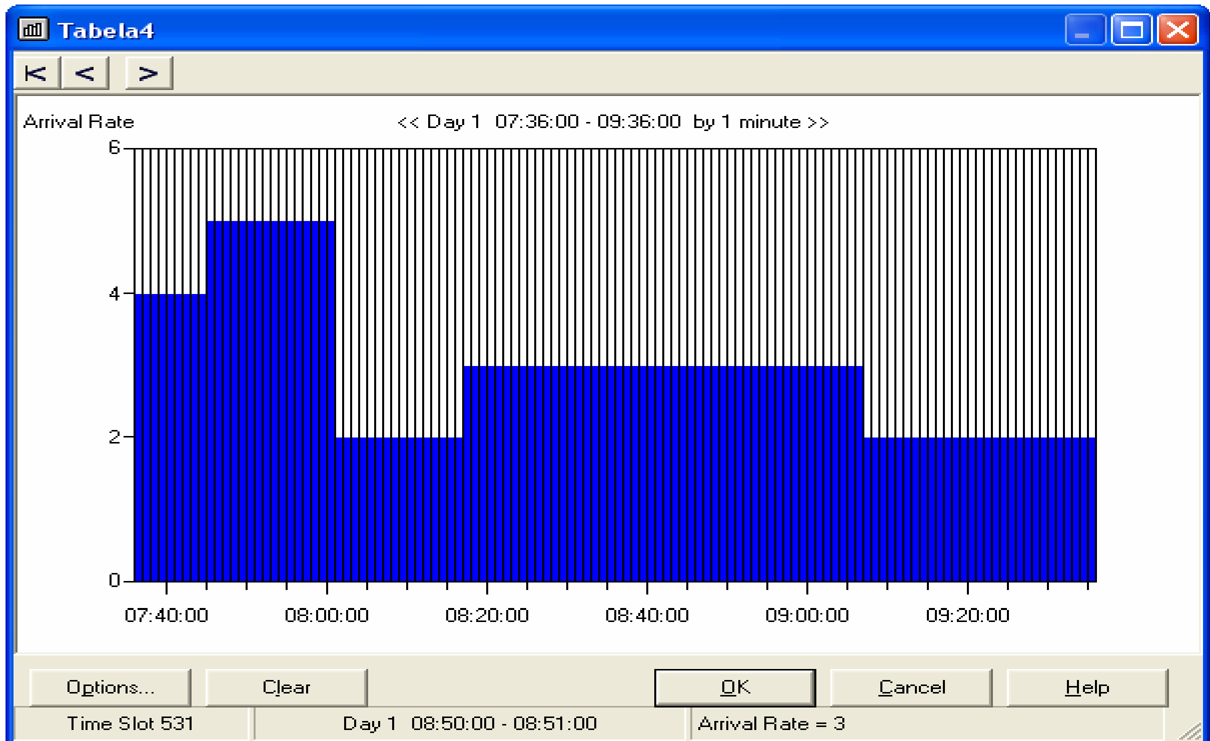

Fig 10. Gráfico de ritmo de chegada.

O módulo *Create* permite a criação de apenas um tipo de entidade por vez. Neste estudo de caso é criada a entidade tipo Carro. Quando o carro é “lançado” no sistema, ele entra primeiramente no bloco Inicialização (tipo *Assign*), o qual é configurado para inicializar os atributos da entidade Carro (ver Fig 11). Os Atributos inicializado são:

- Velocidade Inicial: o Arena disponibiliza vários tipos de distribuição aleatória (Erlang, Gama, Weibull, Beta entre outras) cada uma com sua peculiaridade. Foi utilizada a distribuição Uniforme para atribuição da velocidade inicial. Devido ao conhecimento de apenas dois valores: limite mínimo e limite Maximo. Para gerar as velocidades utilizou-se a expressão: UNIF(0,1)*15+5. UNIF gerar valores entre [0:1] multiplicados por 15 o intervalo aumenta para (0;15) somando a 5, garante que carros nunca entram no sistema com velocidade igual a zero.

- Posição Atual: a inicialização não pode ocorrer com o valor zero, então foi atribuído a esta variável um valor baixo (0.1)

Os outros atributos do tipo *Entity* do Arena são utilizados para fazer o controle da entidade Carro no sistema.

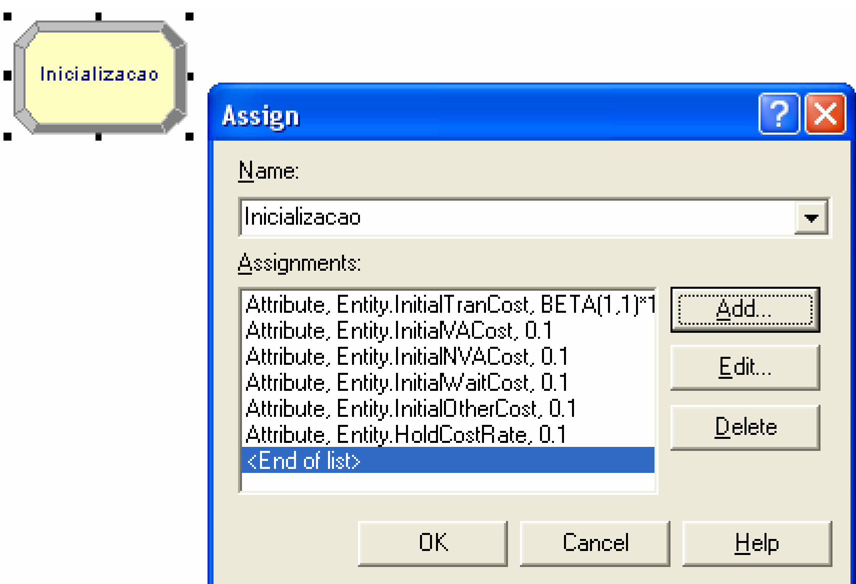

Fig 11. Configuração do bloco inicialização.

b) Modelagem da Estrada

Depois que os carros tiveram seus atributos inicializados, os mesmos entram na estrada para percorrê-la. Nesta modelagem o processo de “andamento em uma estrada” foi representado através de um submodelo, que contém módulos que foram combinados para representar a sua lógica. Foram usados especificamente dois tipos de módulos: o módulo *Decide* (Decisão) do Arena, que verifica se o carro chegou no final da estrada e o módulo *Assign* (Atribuidor) do Arena que calcula o espaço que o carro percorre. A Fig 12 mostra o conteúdo do submodelo Estrada 1.

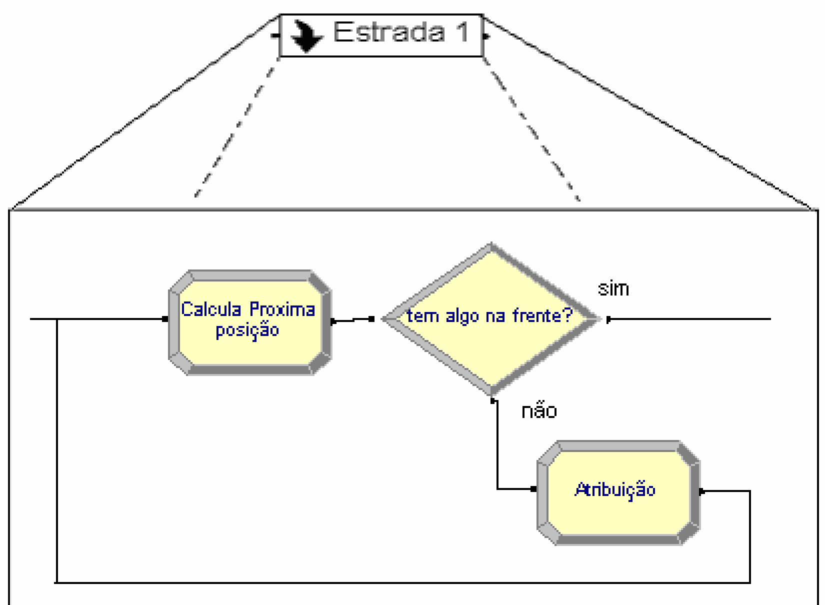

Fig 12. Representação da estrada.

Quando os carros chegam no submodelo Estrada 1 é calculada sua próxima posição. Caso não existe nenhum objeto na frente do carro a posição calculada é atribuída ao carro e uma nova

posição será calculada. Caso existe um objeto na sua frente, mas especificamente um semáforo, o carro segue os passos conforme a Fig 15.

c) Modelagem do Semáforo

Conforme visto no modelo conceitual, os semáforos são entidades estáticas. Não existe um módulo específico que represente a mudança de estado (cor) de um semáforo no Arena, mas existem módulos que podem ser combinados, para representá-la.

As mudanças de cores dos semáforos podem ser representados por submodelos que implementam a sua lógica. O semáforo possui três estados, que são representados pelas cores: verde, amarelo, vermelho essas cores possuem respectivamente os seguintes valores 1, 2 e 3 no sistema modelado. Na Fig 13 mostramos os blocos que foram combinados para se projetar o controle das cores de um semáforo. A mudança de estado de um semáforo é simplesmente a alteração no valor de variáveis que representam a cor do semáforo.

Dentro do submodelo existe um modulo *Create* que, de tempos em tempos, "lança" uma entidade auxiliar dentro do submodelo, essa entidade auxiliar vai para o bloco *Assign* (Atribuído ver Tabela 6), onde o valor 1 (cor verde) é atribuído à variável cor_1 e o valor 3 (cor vermelha) é atribuído à variável cor_2; em seguida a entidade auxiliar é retida no bloco *Process* (Processo) por um tempo pré-determinado por cada cor. Depois de "processada" a entidade dirigi-se a outro bloco *Assign*.

O valor 2 (cor amarela) é atribuído a variável cor_1 e a variável cor_2 continua com 3 e entidade auxiliar desloca-se para outro *Process* onde fica retida por um tempo determinado por cada cor. E assim sucessivamente até chegar ao final e ser retirada do submodelo pelo bloco *Dispose* (Saída). A Fig 14 mostra o processo de atribuição das variáveis.

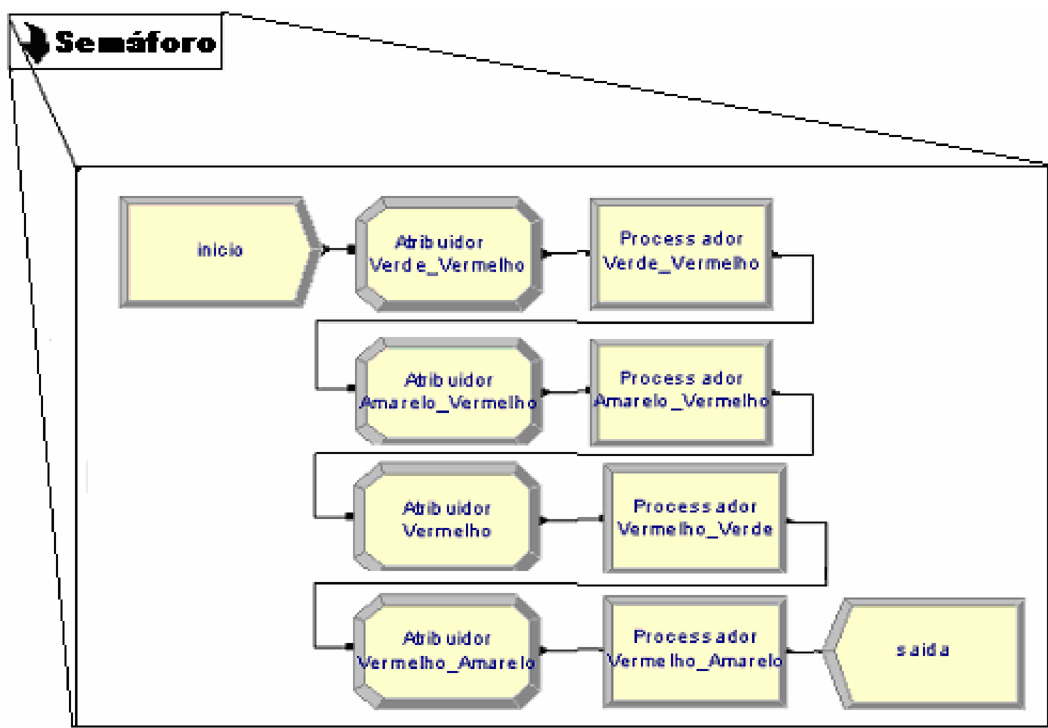

Fig 13. Funcionamento do semáforo.

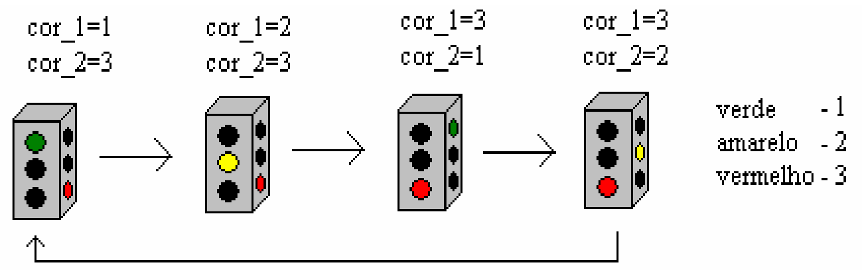

Fig 14. Atribuição de variáveis na mudança de cores.

Quando os carros encontram um semáforo, eles se deparam com um módulo **Tem Fila?** (*Decide*) 1) Se há fila (variável TemCarroNaFila1 é diferente de zero) os carros são direcionados para o bloco **Parar Carro**. Neste bloco os carros recebem velocidade atual igual a zero, a variável TemCarroNaFila1 é incrementado de um e inicia a contagem do tempo em espera na fila. Em seguida os carros se enfileiram no bloco **Fileira** (*Branch*) e permanecem nele até o semáforo "abrir". Quando o semáforo abre os carros são direcionados para bloco **Mover Carro**, no qual os carros recuperam sua velocidadeAtual, a variável TemCarroNaFila1 é decrementada de um, a contagem do tempo de espera é encerrada e os carros seguem seu movimento. 2) Se não há fila os carros são direcionado ao bloco **Sinal Aberto?** Se sim os carros continuam seu movimento normalmente, senão os carros são direcionados para o bloco **Parar Carro** e os passos descritos anteriormente são seguidos. A Fig 15 pode ajudar para entender melhor esta descrição.

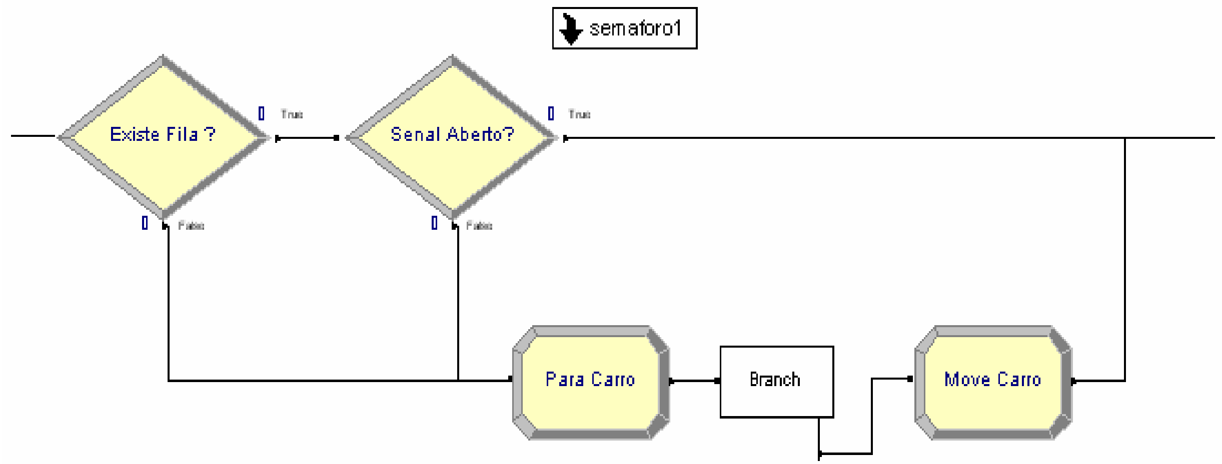

Fig 15. Verifica fila e semáforo.

Finalmente, depois de percorrer toda a estrada os veículos chegam ao final da simulação e encontram o módulo *Dispose* no fim de cada estrada, ver Fig 16, responsável por retirar as entidades e registrar dados relacionados com as estatísticas do sistema.

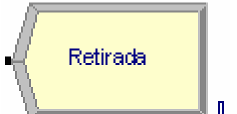

Fig 16. Módulo D*ispose*.

A Fig 17 exibe a parte da Estrada 1 implementada nos passos descritos anteriormente. O modelo completo pode ser visto no Anexo II.

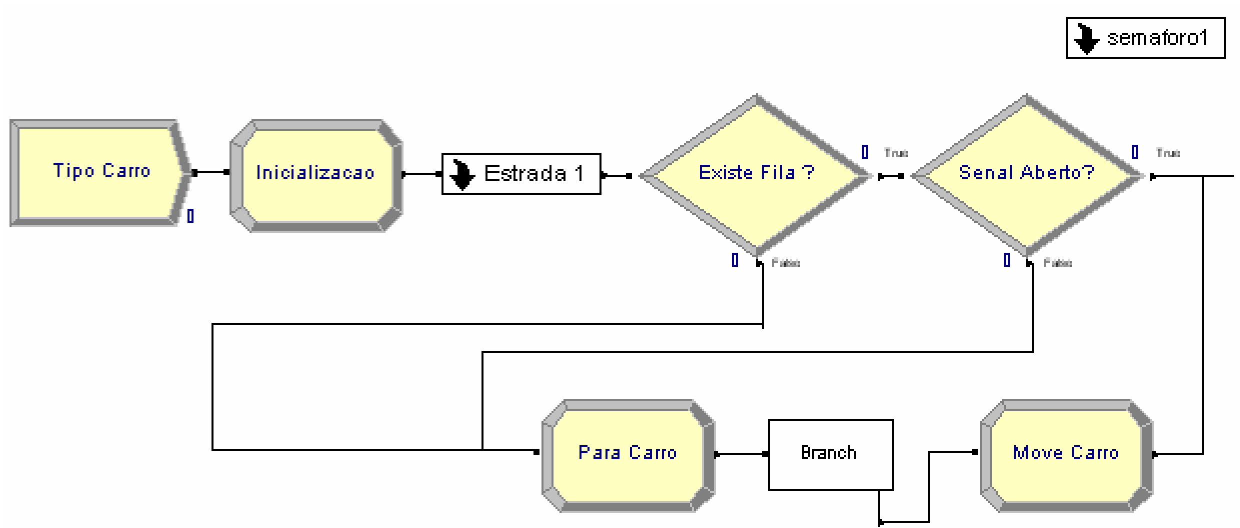

Fig 17. Parte da Estrada 1.

Na Tabela 6 mostramos todos os módulos que foram utilizados na simulação com sua descrição.

Tabela 6. Módulos utilizados na simulação.

Módulos	**Descrição**
Create 1	Responsável por criar as entidades e lançá-las no sistema.
Assign 1	Responsável por fazer atribuições de variáveis, atributos e entidades.
Dispose 1	Responsável por retirar as entidades da simulação e guardar às estatísticas da simulação.
Decide 1	Responsável por decidir o caminho das entidades. Se a concepção for verdadeira segue um caminho senão segue outro caminho.
Branch	Responsável por reter as entidades em fileiras até uma condição ser atingida.

Verificamos que alguns dos objetivos traçados no modelo conceitual foram alcançados. O ambiente de simulação genérico do Arena proporciona várias facilidades para o modelador. Contudo o mesmo precisa conhecer bem o ambiente para saber combinar os módulos disponíveis.

3.2.3.2 Tradução para modelo em Java

Nas linguagens de propósito geral não existem módulos semiprontos, todo controle precisa ser feito pelo programador. O diagrama de classes apresentado na Fig 18 mostra uma visão geral da organização das classes no programa de simulação, além de exibir a multiplicidade entre as mesmas. Um carro percorre apenas uma estrada por vez, enquanto que uma estrada pode ser percorrida por vários carros ao mesmo tempo. Um semáforo pertence a duas estradas e uma estrada possuir 2 semáforos. Os Carros, Estradas e Semáforos são instanciados pela classe **TesteDaSimulacao**, por isto estas classes mantêm uma relação de dependência com ela, representada pela linha potilhada**.** Não são considerados tempos nas filas individuais nem quais carros estão nas filas.

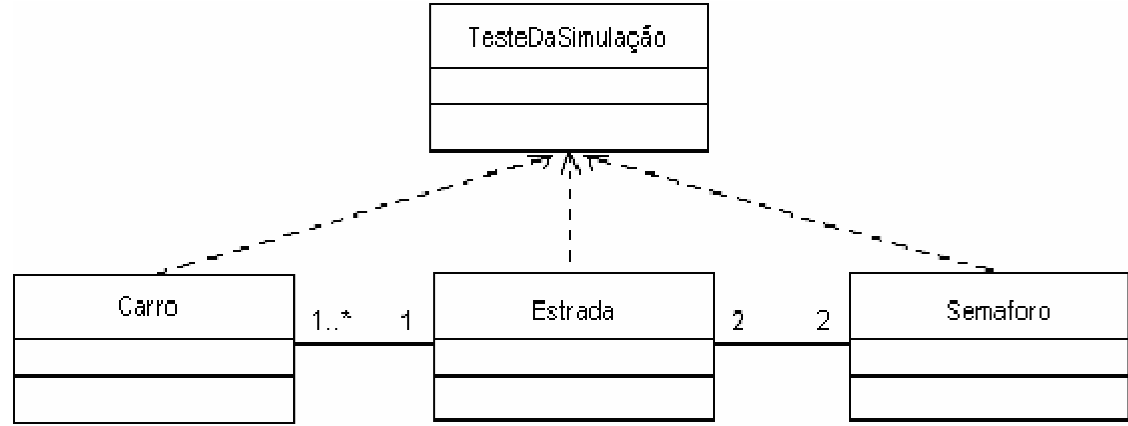

Fig 18. Diagrama de Classes da simulação.

a) Modelagem do Carro

A Fig 19 apresenta a classe Carro modelada em Java, seus atributos e seus métodos. Dentre os atributos destacamos a utilização da **velocidadeInicial** a qual é recebida quando o carro entra no sistema. O atributo **velocidade** guarda a velocidade do carro no momento atual. Destacamos também o **campoVisual**, que é a distância máxima a qual o carro consegue "ver" se existe um objeto na sua frente. Além do mais, temos o atributo estático **placa** que identifica o carro, o atributo **posiçãoAtual** informa o local do mesmo a qualquer momento da simulação.

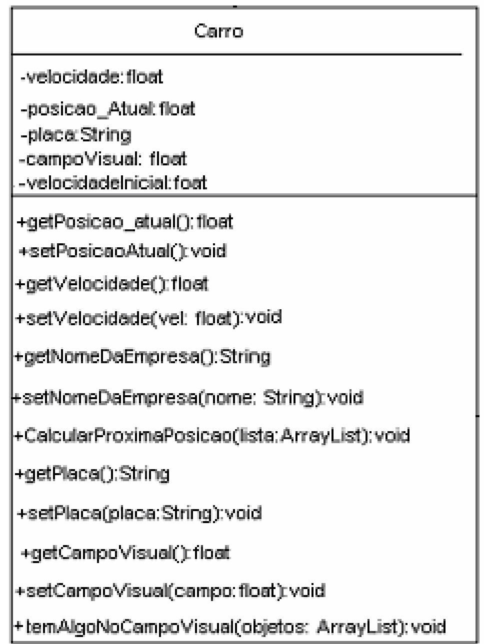

Fig 19. Os atributos e os principais métodos da classe Carro.

Dentre os métodos destacamos o **calcula_próxima_posição()**, responsável por calcular a posição do carro no próximo segundo, também destacamos o método **tem_algo_no_campo_visual(),** responsável por idententificar se existe algum objeto (carro ou semáforo) na frente do veículo e qual é esse objeto (ver Fig 20). Existem métodos *gets*, *sets,* além de outros que auxiliam nos cálculos.

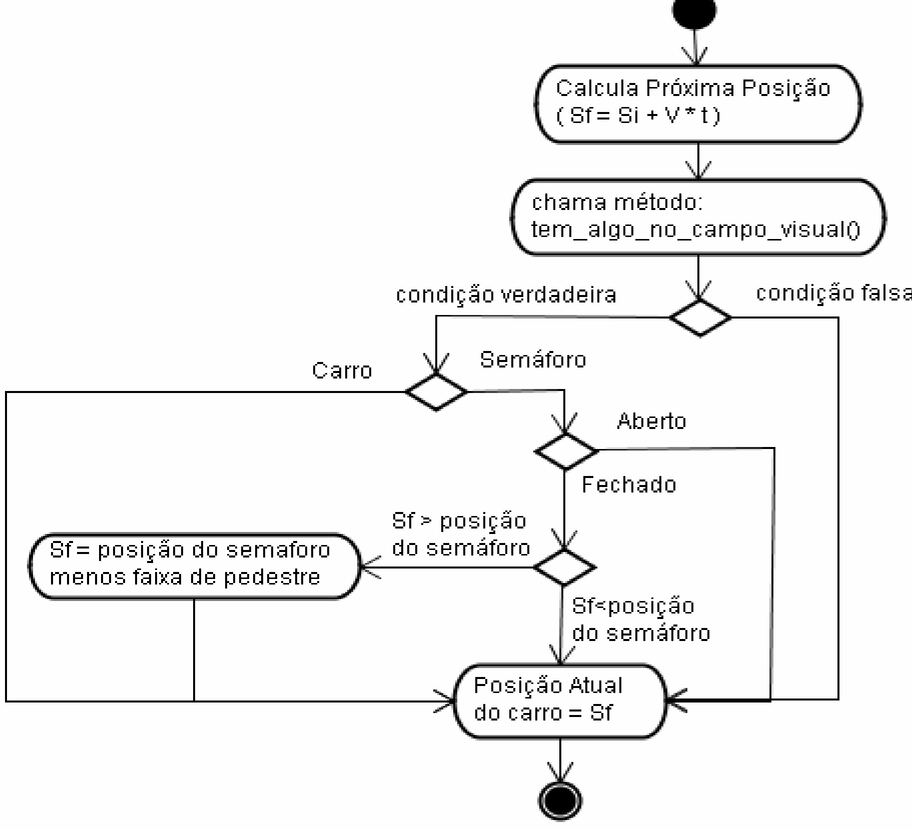

Fig 20. Diagrama de Atividades do Carro.

b) Modelagem da Estrada

A classe Estrada é uma das classes mais centrais, pois os carros circulam por ela e os semáforos localizam-se ao longo do seu comprimento. Portanto, a Fig 21 mostra os atributos e os métodos para entendermos seu funcionamento. Toda estrada possui um comprimento (**comprimento**) e um nome (**nomeDaEstrada**) que é único. Além do mais, as estradas possuem uma lista dos objetos que estão "dentro" dela, ou seja, os veículos e os semáforos que estão localizados nela são guardados no atributo **objetosNaEstrada**.

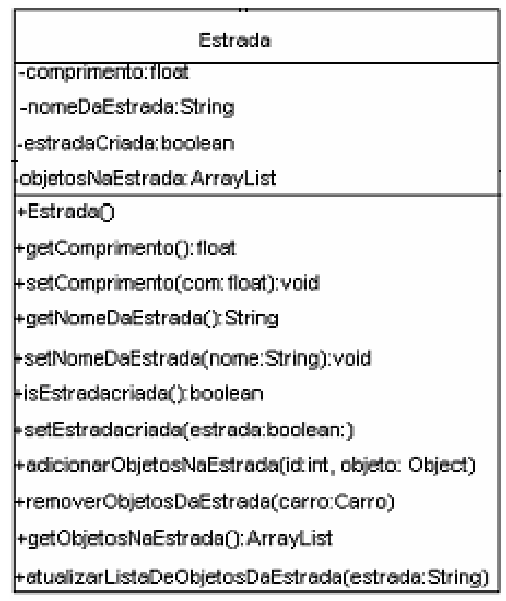

Fig. 21. Atributos e parâmetros da classe Estrada.

O principal método desta classe é o método **atualizar_lista_de_objetos_na_estrada()**, o qual é responsável por verificar se os veículos chegaram ao fim da estrada. A classe Estrada possui outros métodos como **adicionar_objetos_na_estrada()**, **remover_objetos_da_estrada()** dentre outros que auxiliam na execução da simulação. A Fig 22 mostra o funcionamento de **atualizar_lista_de_objetos_na_estrada()**.

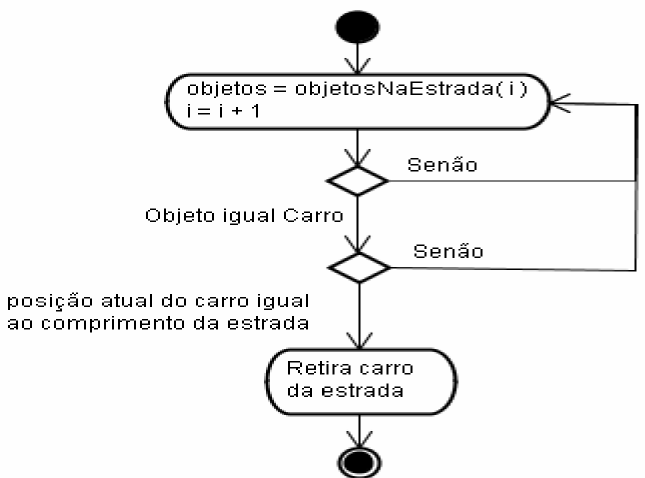

Fig 22. Fluxograma para **atualizar_lista_de_objetos_na_estrada()**.

c) Modelagem do Semáforo

Na classe Semáforo, mostrada na Fig 23, destacam-se as constantes de tempo, que são os tempos que cada cor passa ativada. Os atributos **cor1** e **cor2** informam quais as cores ativadas no momento (para cada sub-semáforo), e os atributos **tempoCor1** e **tempoCor2,** indicam quanto tempo às cores estão ativadas respectivamente. Os atributos **posição_1** e **posição_2** informam as posições do semáforo nas estradas que ele pertence. Dentre os métodos destaca-se o **atualizarTempo()** responsável por atualizar os tempos das cores atualmente ativadas.

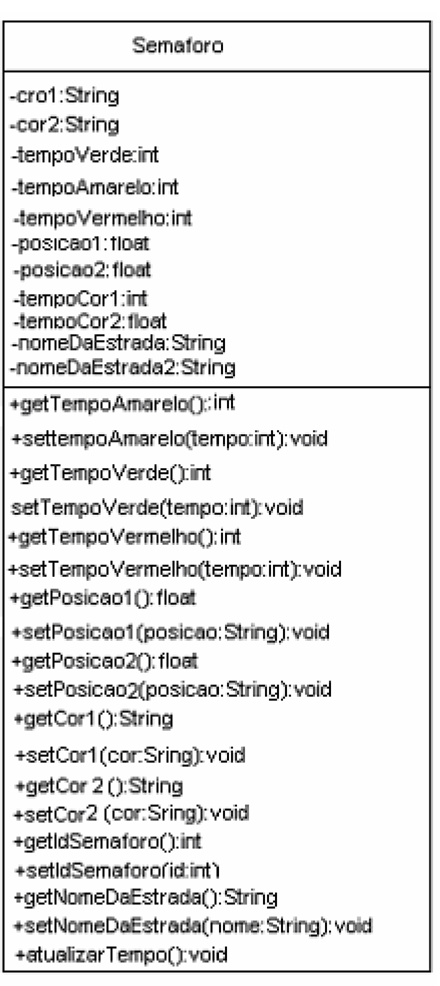

Fig 23. Os atributos e principais métodos do semáforo.

Depois de definidas as classes da simulação criamos a classe principal, chamada **TesteDaSimulação** (ver Fig 24). Esta classe é responsável por controlar o tempo da simulação, criar os objetos, atualizar cada objeto entre outras funções. Destacamos os atributos **tempoNaFilaEstrada1**, **tempoNaFilaEstrada2** e **tempoNaFilaEstrada3** que contam quanto tempos os veículos permanecem parados, destacamos também os atributos **tempoNaEntrada** e **tempoNaSaida** que armazenam respectivamente o tempo que o carro entra na estrada e o tempo que o carro sai da estrada.

Para iniciarmos a simulação precisamos criar as estradas e os semáforos que irão estar localizados nela e interligar as estradas. Conforme o trecho de código contido no método *main*, apresentado na Fig 25, são criadas três estradas chamadas de: Estrada 1 com 1000 m de

comprimento, Estrada 2 com 500 m de comprimento e Estrada 3 com 500 m de comprimento. São criados também dois semáforos. Um localizados na posição 300 da Estrada 1 e na posição 200 da Estrada 2 e outro na posição 700 da Estrada 1 e 300 da Estrada 3. Por fim, as estradas são conectadas através dos semáforos.

TesteDaSimulacao
+tempoNaEntreda:HashMap +tempoNaSaida:HashMap +tempoNafilaEstrada1:int +tempoNafilaestrada2:int +tempoNaFilaEstrada3:int +totaldeCarrosEstrada1:int +totalDeCarrosEstrada2:int +totalDeCarrosEstrada3:int
+gerarVeiculos(hora:int, nomeEstrada:String) main(String args)

Fig 24. Classe TesteDaSimuação.

```
        Estrada estrada1 = new Estrada(“Estrada 1”, 1000);
        Estrada estrada2 = new Estrada(“Estrada 2”, 500);
        Estrada estrada3 = new Estrada(“Estrada 3”, 700);
        Semáforo semaforo1 = new Semaforo("verde",“vermelho”,300, 200);
        Semáforo semaforo2 = new Semaforo("vermelho",“verde”,700, 300);
if(estrada1.isEstradaCriada()){
        estrada1.adicionar(semaforo1);
        estrada1.adicionar(semaforo2);
}
if(estrada2.isEstradaCriada()){
        estrada2.adicionar(semaforo1);
        }
if(estrada3.isEstradaCriada()){
        estrada3.adicionar(semaforo2);
}
```

Fig 25. Trecho de código de criação de três estradas, dois semáforos e conexão das estradas.

Depois das estradas e dos semáforos terem sido criados, e os semáforos serem interligados às estradas, inicia-se a simulação. O tempo de simulação é controlado por um laço, e a velocidade de execução é controlado por outro laço. O código da Fig 26 mostra estes dois laços.

```
for(int hora=0; hora < 24; hora++){//equivalente a 24 horas de simulação
        for(int segundos =1; segundos<=3600; segundos++ ){ //quant. de seg. em 1 hora
                for(int retardo=0; retardo<1000000; retardo++); //velocidade de execução
```

Fig 26. Tempo de simulação e velocidade de execução.

A Fig 26, o tempo de simulação é de 24 horas e cada ação é realizada de segundo em segundo, pois em uma hora a 3600 segundos. O laço mais interno controla a velocidade de execução, para a execução ser mais rápida só é diminuir o número de interações, para torná-la mais lenta só é aumentar o número de interações.

Os carros entram no sistema através do método **criarVeiculos()**, o qual obedecendo ao fluxo de carros apresentado nas Tabela 3, 4 e 5. Ele "escolhe" aleatoriamente a quantidade de carros inseridos no sistema. O trecho de código da Fig 27 (baseado na Tabela 3) exibe que para horários entre 6:00 e 8:00 horas, 12:00 e 14:00 horas e 18:00 e 20:00 horas, a quantidade de carros que entra no sistema está no intervalo de 10 a 14 carros.

```
if((hora>=6 && hora < 8) || (hora>=12 && hora < 14) ||  (hora>=18 && hora < 20)){
        ritmoDeChegada = 10 + (int) (Math.random() * 5); // escolhe num entre 0 - 4
}
```

Fig 27. Trecho de código do método criarVeículo().

A simulação faz uma atualização dos objetos de segundo a segundo, ou seja, se o tempo está em T, o mesmo passará a T +1 quando todos os objetos do sistema atualizarem seus estados. O código das Fig 28 mostra a estrada identificando os objetos e atualizando os mesmos.

```
//pega objetos da estrada1
ArrayList objetos = estrada1.getObjetosNaEstrada();
Iterator it = objetos.iterator();

while(it.hasNext()){
Object obj = it.next();

if(obj instanceof Semaforo){//obj Semaforo
        Semaforo sem = (Semaforo) obj;
        sem.atualizarTempo(); //atualiza
}
        else if (obj instanceof Carro){//obj Carro
        Carro car = (Carro) obj;
        //atualiza posição do Carro
        car.calculaProximaPosicao(estrada.getObjetosNaEstrada());
}

}//fecha while(it…
```

Fig 28. Código da atualização dos objetos.

Depois de todos os objetos atualizarem seus estados atuais, a simulação volta ao começo. Verifica que horas são, e cria a quantidade de carros de forma aleatória para o horário determinado de acordo com as tabelas 3, 4 e 5. Todos os objetos são chamados a atualizar seus estados novamente, e assim por diante até terminar o tempo de simulação. O código completo da modelagem para essa simulação pode ser visto no Anexo III. Em seguida a Fig 28 mostra o diagrama de classes da simulação. Este diagrama mostra a interação entre os objetos que compõe a simulação.

3.2.4 Verificação e Validação

Nesta etapa verifica se o programa de computador executa o modelo da forma esperada e a validação é a confirmação de que um modelo é uma representação adequada do sistema real. Durante a verificação foi observado se as implementações eram adequadas para o modelo do sistema e se funcionava adequadamente. O propósito da verificação é assegurar se o modelo conceitual seja refletido adequadamente na representação computacional [1]. Banks [1] apresenta algumas sugestões que podem ser usadas no processo de verificação. Entre elas foram usadas:

- Representação examinada pela animação, visto que ela torna mais fácil a visualização dos erros.
- Examinar as saídas do modelo, para verificar se elas são razoáveis para os parâmetros de entradas.
- Utilização de interface gráfica (tabelas, gráficos, animações, ect.) para avaliar se o comportamento das variáveis estava correto.

A validação do modelo programado muitas vezes é realizada comparando-se os dados obtidos no modelo com os dados do sistema. Essa técnica não pode ser usada neste trabalho; já que o sistema simulado é hipotético, foi utilizado uma das técnicas descritas em [11] como: coleta de informações e dados de alta qualidade do sistema para fazer uma validação do modelo desenvolvido. A Coleta de informação e dados de alta qualidade refere-se a coleta de toda informação existente sobre o sistema. Isso foi realizado por meio da análise da teoria existente e da intuição do modelador e manutenção de um documento de suposições e desempenho.

3.3 Execução do Modelo

As tabelas 3, 4 e 5 foram utilizadas para a análise do sistema, durante o período de um dia completo, ou seja, 24 horas. Foram variados os tempos de percursos e os tempos de ativação de cada cor no sistema, permitindo assim respostas mais confiáveis do modelo.

3.3.1 Projeto Experimental e Análise dos Resultados

Nesta etapa são determinadas as alternativas que serão usadas no estudo da simulação e a estratégia utilizada. Isso inclui as variáveis de respostas usadas, quais os fatores (variáveis que afetam as variáveis de resposta) e os níveis (valores que os fatores podem assumir) a serem considerados, o número de replicações, etc. Além da estratégia utilizada (tipo de projeto experimental, por exemplo, projeto simples fazendo variar um fator por vez).

Os fatores utilizados neste trabalho foram: o tempo de ativação dos semáforos e o comprimento das estradas. Os níveis são apresentados na Tabela 7. A estratégia utilizada foi a alteração de um fator por vez descrita em [5]. A combinação de todos os fatores pode ser vista na Tabela 12, no Anexo V. A unidade de tempo escolhida foi o segundo, o que proporcionou uma maior representação dos detalhes do modelo. Foi escolhido o intervalo de 5 dias (totalizando 432000 segundos) como o tempo total de simulação para cada situação. Ao final deste período foram totalizados os resultados da operação do sistema.

Tabela 7. Parâmetros estudados no modelo.

Parâmetros	**Variações**	
Tempo de ativação dos semáforos (min)		
Verde	4	2
Amarelo	2	1
Vermelho	6	3
Comprimento das estradas (m)		
Estrada 1	1000	500
Estrada 2	500	250
Estrada 3	700	350

As variáveis de respostas que serão utilizadas para medir o desempenho do sistema com as diferentes configurações são:

- Quantidade de carros que passaram pelo sistema.
- Tempo médio que os veículos passaram em cada fila de semáforo fechado de cada estrada.
- Tempo médio que os veículos passaram no sistema.

Os testes realizados objetivaram avaliar a melhor forma de programação dos semáforos na simulação, além de validar os modelos implementados pelas duas ferramentas. As Tabelas 8 e 9 mostram os valores obtidos por 8 casos testados no Arena e no Java. Como o restante dos valores apresenta um padrão de comportamento similar optou-se por omití-los aqui. As Tabelas 8 e 9 podem serem vistas completas no Anexo V. Os modelos são as diferentes variações dos fatores. No anexo V a tabela 12 mostra todas as combinações dos fatores utilizados.

Tabela 8. Resultados dos testes no Arena.

Modelo	Tempo no sistema (minuto)	Tempo na fila (minuto)	Quantidade de entidades		
			Estrada 1	Estrada 2	Estrada 3
1	11	4	1074	1379	1259
2	8	4	1115	1403	1278
3	10	4	1074	1397	1259
4	7	4	1115	1411	1278
5	10	4	1074	1379	1280
6	8	5	1111	1405	1250
7	9	3	1074	1397	1280
8	7	3	1115	1411	1258
...	...	...	...	...	...

Tabela 9. Resultados dos testes no Java.

Modelo	Tempo no sistema (minuto)	Tempo na fila (minuto)	Quantidade de entidades		
			Estrada 1	Estrada 2	Estrada 3
1	13	6	1324	1639	1509
2	9	4	1375	1663	1528
3	12	5	1314	1637	1499
4	10	3	1365	1651	1518
5	10	4	1334	1694	1574
6	7	4	1371	1665	1491
7	8	4	1304	1627	1510
8	7	3	1355	1661	1508
...	..	..	...	..	..

Primeiramente observa-se que os valores obtidos pelos modelos, no Arena e em Java, são bastante semelhantes. Concluímos que independente do tipo de implementação a simulação produz resultados parecidos. Podemos tirar essa conclusão em função dos níveis de detalhes serem praticamente os mesmo em ambas as implementações. Portanto na escolha de uma ferramenta dentre tantas outras que produzem os mesmos níveis de detalhes deve-se escolher, primeiramente aquela conhecida pelo desenvolvedor e em seguida a que fornece as maiores facilidades de desenvolvimento.

Outro ponto a ser analisado é o tempo de ativação dos semáforos. Observa-se que diminuindo o tempo de processamento dos semáforos ou seja, diminuindo o tempo de ativação de cada cor, provoca-se um menor tempo das entidades no sistema e um aumento na quantidade de carros que passam pelo sistema, em ambos os modelos acontecem isso. Assim concluímos que os semáforos passavam muito tempo ativado numa cor bloqueando a passagem de veículos por outras estradas.

A Tabela 10 faz uma comparação entre o sistema real, o modelo conceitual e o que foi implementado nas duas abordagens que este trabalho se propôs a implementar e analisar. Um terceiro ponto a ser analisado é que diminuindo o comprimento das estradas provoca uma diminuição no tempo da entidade e aumenta a quantidade de veículos que passam pelo sistema, mas de forma mais tímida que a mudança nos tempos dos semáforos. Portanto a combinação destes dois pontos: tempo de ativação dos semáforos e tempo de transposição provocaram os menores tempos de permanência dos sistemas das entidades.

As conclusões e observações aqui levantadas respondem por apenas uma pequena parte das análises que se poderia fazer a respeito do sistema; considerando-se que o caso aqui abordado é simples, pode-se concluir que a simulação oferece uma diversidade de meios e ferramentas para análise de cada ponto crítico encontrado em sistemas reais. O estudo aqui realizado buscou apenas mostrar o caminho a ser trilhado até que conclusões consistentes e importantes possam ser tiradas.

Tabela 10. Análise do sistema implementado.

Sistema Real	Modelo Conceitual	Tradução Arena	Tradução Java
Criação de três estradas com comprimento e largura.	Criação de três estradas apenas com comprimento	Implementado	Implementado
Semáforos com dois conjuntos de três cores sincronizadas no mesmo conjunto e entre conjunto.	Semáforos com dois conjuntos de três cores sincronizadas no mesmo conjunto e entre conjunto.	Implementado	Implementado
Veículo entra nas estradas com velocidade inicial.	Veículo criado com velocidade inicial.	Implementado	Implementado
O ritmo de chegada do carro depende do horário.	Ritmo de chegada de acordo com as Tabelas 3, 4 e 5.	Implementado	Implementado
Veículos com campo visual	Veiculo “enxergar” o que existe na sua frente até uma certa distância.	Implementado	Implementado
Os carros formam uma fila quando encontram o semáforo fechado.	Carros se enfileiram quando encontram um semáforo fechado.	Implementado	Implementado
Os carros mais velozes ultrapassam os carros mais lentos	Admite que os carros ultrapassam os mais lentos sem nenhuma preocupação com a largura da estrada ou demissões do veículo.	Implementado	Implementado

3.5 Considerações finais

As seções anteriores mostraram o desenvolvimento de um estudo de caso em duas abordagens diferentes usando para isso o Arena e o Java. O desenvolvimento do modelo apresentado na sessão mostrou que existem facilidades e dificuldades em ambas as abordagens. Apesar do modelo implementado ser simples, devido à existência de poucos processos lógicos para efetuar a simulação, procurou-se mostrar as potencialidades das duas abordagens, sem favorecer e nem desmerecer nenhuma das duas.

A Tabela 11 mostra algumas características observadas durante o desenvolvimento do estudo de caso. Observamos que os ambientes de simulação apresentam várias facilidades para o programador na hora de traduzir o modelo, contudo o programador fica um pouco preso à estrutura daquela ferramenta. Toda ferramenta de programação seja ela um ambiente de simulação genérico ou específico, ou de propósito geral é uma visão de mundo dos seus

desenvolvedores, portanto o programador fica preso àquela visão de mundo implementada na ferramenta.

Apesar das linguagens de propósito geral também apresentarem certa visão de mundo, e fornecerem estruturas fixas que prendem o programador, elas fornecem uma maior liberdade que os ambientes de simulação e um maior controle do sistema. Por outro lado, o programador fica com a responsabilidade de implementar tudo de que ele vai precisar, tornando assim a implementação da simulação uma tarefa muito mais árdua, e em alguns casos complexa, fazendo o tempo de desenvolvimento ser maior.

De uma maneira geral a vantagem de se usar um ambiente de simulação genérico está diretamente ligada a desvantagem de usar uma linguagem de propósito geral, e vice-versa [18]; as características que são fortes em uma ferramenta são os pontos fracos da outra. Assim um aumento na flexibilidade de um programa de simulação é obtido à custa de um estudo mais aprofundado dos processos de programação. A Tabela 11 resume em termos as características analisadas na modelagem do estudo de caso nas duas abordagens.

Tabela 11. Atributo de Qualidade.

Características analisadas	**Ambiente Genérico**	**Linguagem de propósito geral**
Disponibilização de diferentes lógicas de processamento (módulos de controle)	+ Fácil grande quantidade de módulos (ex: Create, Branch, Decide, etc)	Tem de ser codificado
Controle e acompanhamento de estatísticas	+ Fácil	Tem de ser codificado
Realização de animações	+ Fácil	Tem de ser codificado
Controle de recursos, entidades	+ Fácil	Tem de ser codificado
Controle da execução de eventos e tempo de simulação	+ Fácil	Tem de ser codificado
Flexibilidade na implementação	Menor	Maior
Desenvolvimento	+ Rápido	+ Demorado
Curva de aprendizado	+ lenta	+ Rápida
Realização de experimentos e otimização	+ Fácil	+ Trabalhosa
Tratamento de dados (E/S)	+ Fácil	+ Trabalhosa
Manutenção de modelos	+ Fácil	+ Difícil

Por este trabalho se tratar de um projeto de conclusão de curso, o estudo de caso sofreu uma série de simplificações tendo em vista cumprir um conjunto de exigências acadêmicas dentre elas o prazo de conclusão do trabalho.

Não foi possível trabalhar com todas as variáveis do sistema e nem com todas as situações, como por exemplo, ultrapassagem de carros. Mas, todavia, o trabalho desenvolvido, apesar dos cortes, traz consigo o êxito de obter maior conhecimento sobre os sistemas de simulação e de tráfego.

Capítulo 4

Conclusões e Propostas para Trabalhos Futuros

4.1 Conclusões

A simulação tem sido utilizada em nas mais diversas áreas, por conta disso foram produzidas durante os últimos temos várias ferramentas de simulação visando obter uma maior eficiência dos modelos simulados, diminuir custos e produzir projetos em maior velocidade. Escolher a ferramenta mais adequada, para um dado problema pode fazer grande diferença.

O presente trabalho permitiu estabelecer uma série de estudos sobre simulação, assim como tirar conclusões pertinentes do ponto de vista da escolha da ferramenta de simulação adequada para o desenvolvimento de um modelo de simulação.

Após o desenvolvimento e experimentação nas duas ferramentas, observou-se nos resultados das simulações que tanto no Arena como no Java existia uma proximidade nos valores encontrados. Apesar da simplicidade do estudo de caso e das simplificações introduzidas em cada modelo, pode-se concluir que o modelo desenvolvido no ambiente de simulação genérico e o modelo desenvolvido na linguagem de propósito geral são, praticamente, os mesmos. Então pode surgir o questionamento: “Poderia sido usado qualquer uma das duas ferramentas para desenvolver uma simulação?”. Praticamente sim. Contudo, foi visto ao longo do trabalho que os ambientes de simulação apresentam uma maior facilidade para o modelador do sistema, reduzindo o tempo de desenvolvimento. Nestes ambientes existem vários módulos de controle já desenvolvidos e com diversos recursos que ajudam o desenvolvedor, permitindo aos mesmos se concentrarem em outros problemas. Estes sistemas também proporcionam um modo de desenvolvimento bem agradável, visto que quase tudo pode ser feito utilizando apenas o mouse. Mas por outro lado o desenvolvedor fica obrigado a estudar ambientes de programação complexos, pouco divulgados (fora da área de simulação). Os ambientes são restritos a uma visão de mundo implementada pela ferramenta, tendo assim de enfrentar um grau grande de complexidade na programação e tirando assim um pouco sua liberdade de criação. Por outro lado, vimos no desenvolvimento deste trabalho que as linguagens de propósito geral podem também

serem usadas para o desenvolvimento de simulações, permitindo uma grande liberdade de criação aos desenvolvedores. Todavia essas ferramentas implementam a filosofia de "*it-do-youself*", ou seja, faça você mesmo. Dessa forma elas são muito mais trabalhosas e demoradas. Além de ser mais difícil em virtude do desenvolvedor ter um pré-conhecimento da linguagem, ao contrário dos ambientes genéricos, que são mais intuitivos.

O objetivo principal deste trabalho foi analisar as diferenças existentes entre os ambientes de simulação genéricos e as linguagens de propósito para o desenvolvimento de simulações. Todos os objetivos foram alcançados: uma aplicação prática da simulação computacional foi realizada, com o objetivo de aprofundar os conceitos estudados além de enriquecer o trabalho; uma revisão bibliográfica sobre a simulação foi apresentada; aprendizado sobre o Arena foi descrito durante o texto. Nesta ocasião, o ponto principal não foi encontrar resultados e nem respostas para algumas perguntas sobre o sistema, mais sim a análise sobre os passos do desenvolvimento do sistema, as características de cada ferramenta que foram apresentadas. As vantagens e desvantagens de cada abordagem.

Deve-se ressaltar que, para ser uma técnica útil o desenvolvedor deve escolher adequadamente a ferramenta para representar o modelo a ser desenvolvido, a escolha errada faz-se demorar mais a modelagem. Também os passos do estudo devem ser seguidos sistematicamente, os dados devem ser analisados com cuidado e a implementação deve ser conduzida corretamente.

Assim sendo, conclui-se que ambas as ferramentas utilizadas para implementar o modelo de tráfego de automóveis possuem vantagens e desvantagens, que o modelador responsável deve medir antes de decidir qual delas utilizar. Contudo a escolha correta tornará o processo de confecção menos demorado, permitindo ao desenvolvedor aprofundar-se mais nas características do sistema modelado.

4.2 Trabalhos Futuros

Como sugestões para trabalhos futuros os seguintes tópicos podem ser abordados:

- Estudar o ambiente de simulação genérico Arena mais detalhadamente. E desenvolver um estudo de caso bem elaborado com maior complexidade, onde possa extrair as principais potencialidades desse ambiente.
- Estudar os testes de verificação sugeridos por LAW e KERNEL em [11] para sistema que apresentam coleta de dados de forma fictícia. Produzindo um material mais acessível e de fácil entendimento.
- Estudar os ambientes Java de simulação seqüencial e simulação distribuída como JSim, SimJava e etc.
- Estudar a simulação *web* e os ambientes de simulação para *web* desenvolvidos em Java.
- Integração com HLA.
- Reorganização de código Java considerando o conceito de simulação e seus aspectos.

Bibliografia

[1] BANKS, J. *Handbook of Simulation – Principles, Methodology, Applications and Practice*. Atlanta: Jhon Wiley & Sons, 1998.

[2] CAROTHERS, C. D.; FUJIMOTO, R. *Efficient Execution of Time Warp Programs on Het-erogeneous, NOW Platforms. IEEE Transactions on Parallel and Distributed Systems,* 2000.

[3] DAHMANN J. S.; FUJIMOTO, R. M.; WEATTHERLY, R. M. *The departament of defense high level architecture. Winter Simulation Conference*, 1997.

[4] EMSHOFF, J. R.; SISSON, R. L. *Design and use of computer simulation models*. New York: MacMillan, 1970.

[5] FREITAS, P. J. *Introdução a modelagem e Simulação de Sistemas*. Santa Catarina: Visual Books, 1990.

[6] FUJIMOTO, R. M. *Parallel and Distributed Simulation System*. Jhon Wiley & Sons, 2000.

[7] HUSSAIN, A.; KAPOOR, A.; HEIDEMANN, J. *The Effect of Detail on Ethernet Simulation Proceedings of the 18th Workshop on Paralled and Distributed Simulatio*n, 2004.

[8] KHEIR, N. A. *System modeling and computer simulation. 2 ed*. New York: Marcel Dekker, 1996.

[9] KUHL, F.; WEATHERLY, R.; DAHMANN, J. *Creating Computer Simulation Systems An Intruduction to the High Level Architecture*. The MITRE Comporation, published by Prentice Hall PTR, 2000.

[10] KULJIS, J.; PAUL, R. J. *A review of web based simulation: whither we wander*, *Winther Simulation Conference*, 2000.

[11] LAW, A. M.; KELTON, W. D. *Simulation modeling and analysis*. 3. ed. Boston McGraw-Hill, 2000.

[12] LOW, Y.; LIM, C.; CAI, W.; HUANG, S.; HSU, W.; JAIN, S.; TURNER, S. J. *Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation.*1999.

[13] NAYLOR, T. H. *Técnica de simulação em computadores*. São Paulo: Editora Vozes, 1971.

[14] OZAKI, A; FURUICHI, M; TAKAHASHI, K; MATSUKAWA, H. *Design and Implementationof Parallel and Distributed Wargame Simulation System,* 2001.

[15] PEGDEN, C. D.; SHANNON, R. E.; SADOWSKI, R. P. *Introduction to Simulation Using SIMAN*. 2 ed. New York: McGraw-Hill, 1990.

[16] PORTUGAL, L. S. *Simulação de tráfego: conceitos de técnicas de modelagem*. Rio de Janeiro: Editora Inteligência, 2005.

[17] PRADO, D. S. *Usando o Arena em Simulação*. Belo Horizonte: INDG Tecnologia e Serviços, 2004.

[18] SHANNON, R. E. *Systems simulation: the art end science. Englewood Cliffs*, N. J: Prentice-Hall, 1975.

[19] SOARES, L. F. G. Modelagem e Simulação discreta de Sistemas. Editora Campos – LTDA,1992.

[20] WATSON, H.; BLACKSTONE, JR. J. H. *Computer simulation*. 2 ed New York: John Wiley & Sons, 1981.

[21] ZHOU, S; TURNER, S. J.; CAI, W.; ZHAO, H.; PANG, X. *A Utility Model for Timely State Update in Distributed Wargame Simulations*. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.

[22] Disponível em http://java.sun.com (visitado pela última vez em 18 de maio de 2006)

Anexo I

Gráficos dos ritmos de chegada das tabelas 3, 4 e 5

Apresentaremos aqui partes dos gráficos dos ritmos de chegada das estradas 1, 2 e 3 baseados respectivamente nas tabelas 3, 4 e 5.

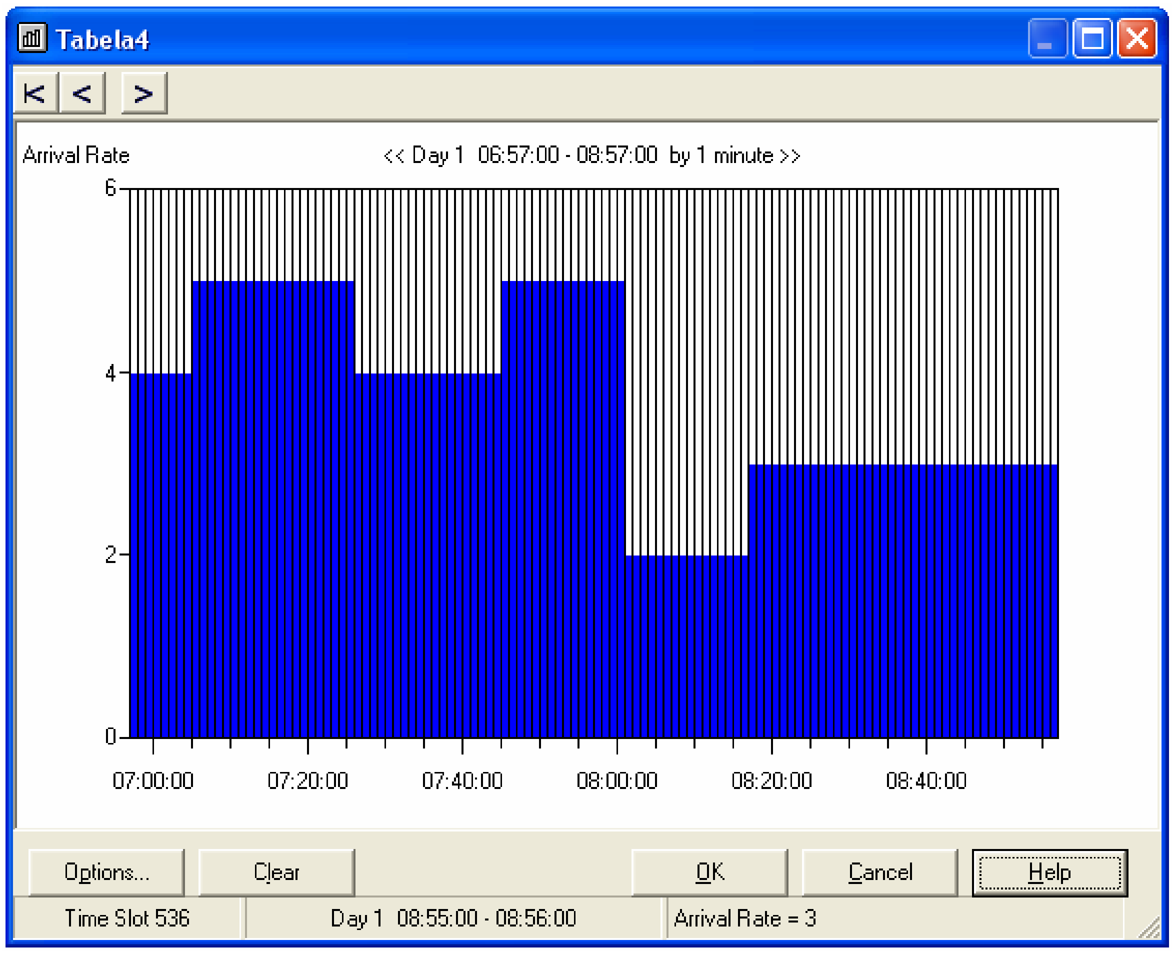

Gráfico de ritmo de chegada da Tabela 3

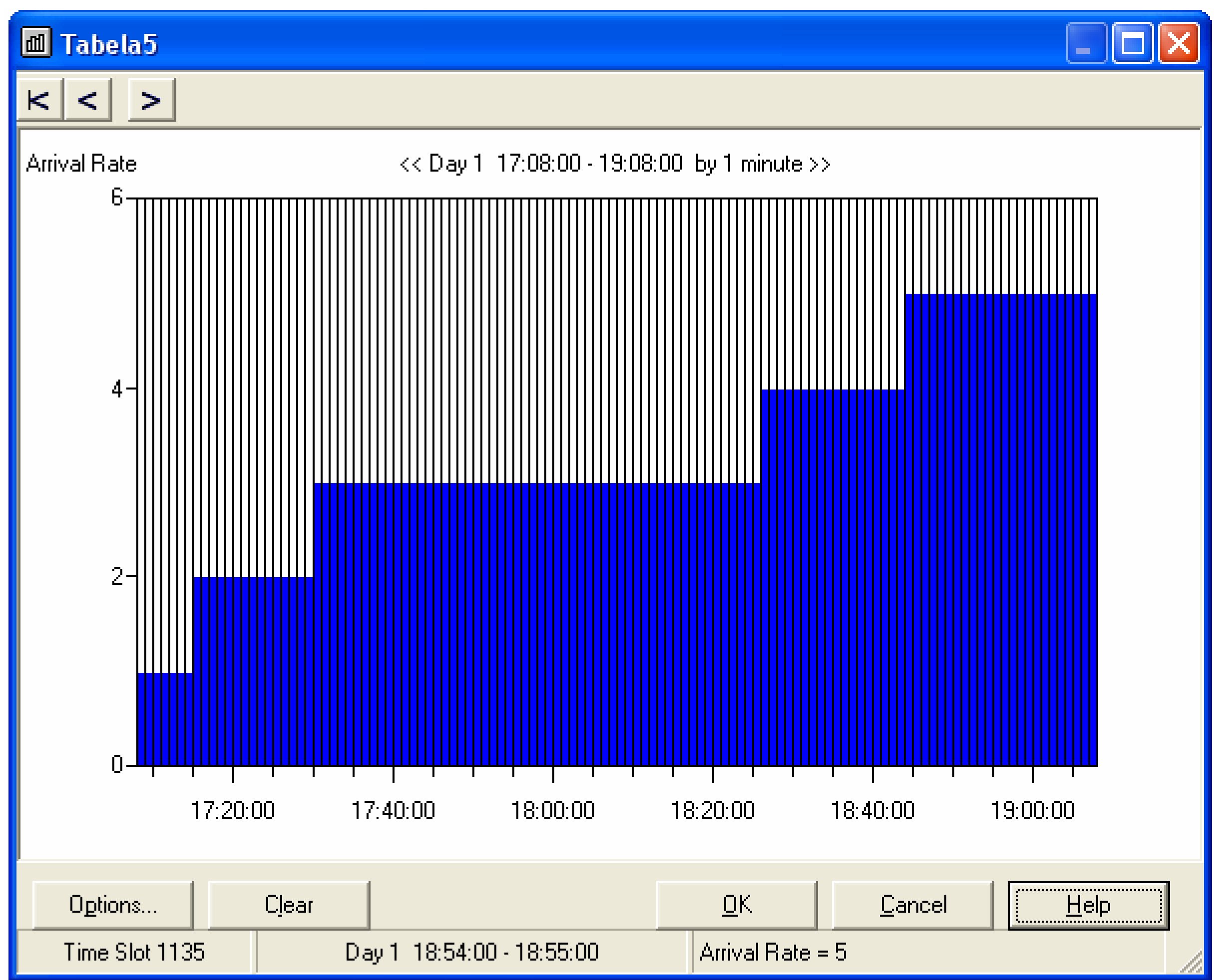

Gráfico de ritmo de chegada da Tabela 4

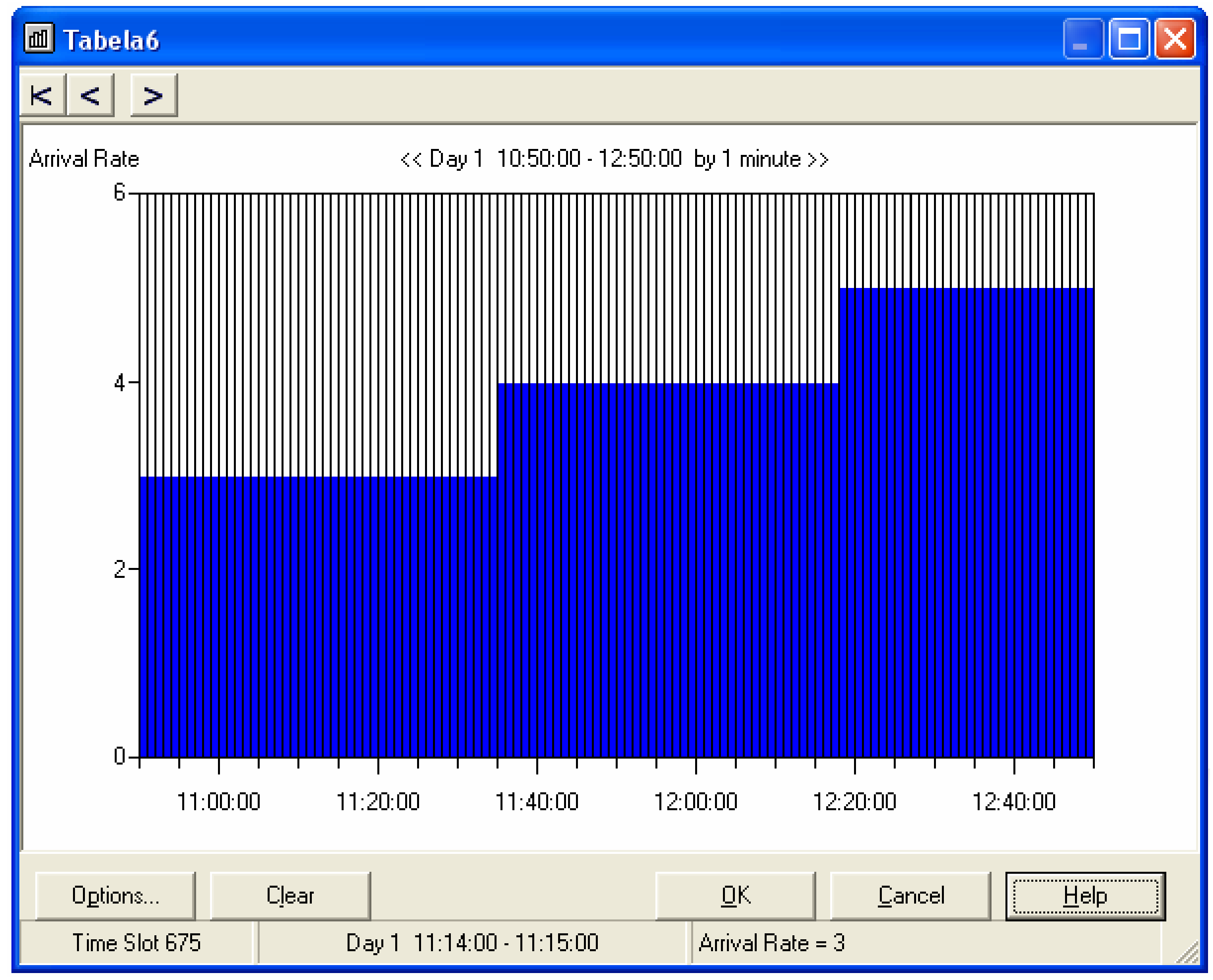

Gráfico de ritmo de chegada da Tabela 5

Anexo II

Modelagem do sistema no Arena

Apresentamos neste anexo o modelo completo do sistema simulado, que se encontram divididos em três partes, para facilitar a visibilidade dos módulos.

Anexo III

Código Java da simulação

```
/**
 * Classe Carro
 */

/*
 * Created on 14/03/2006
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package Basico;

import java.util.ArrayList;
import java.util.Iterator;

/**
 * @author Frédick
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Carro {

	private float velocidade;
	private float posicao_atual;
	private int   idCarro;
	private float campoVisual;
	private float velocidadeInicial;
```

```
	private String nomeDaEstrada;

	public Carro(float vel, float poAt, int id, float caVi, String nomeDaEstrada){
		this.setVelocidade(vel);	//em m/s
		this.setPosicao_atual(poAt);//em metro
		this.setIdCarro(id);
		this.setCampoVisual(caVi);  //em metro
		this.velocidadeInicial = vel;
		this.setNomeDaEstrada(nomeDaEstrada);

	}


	public float getPosicao_atual() {
		return posicao_atual;
	}

	public void setPosicao_atual(float posicao_atual) {
		this.posicao_atual = posicao_atual;
	}

	public float getVelocidade() {
		return velocidade;
	}

	public void setVelocidade(float velocidade) {
		this.velocidade = velocidade;
	}

	public float getVelocidadeInicial(){
		return velocidadeInicial;
	}

	public String getNomeDaEstrada() {
		return nomeDaEstrada;
	}

	public void setNomeDaEstrada(String nomeDaEstrada) {
		this.nomeDaEstrada = nomeDaEstrada;
	}

	public void calculaProximaPosicao(ArrayList objetosNaEstrada){

		//apenas os objetos no campoVisual
		ArrayList objetosNoCampoVisual =
this.temAlgoNoCampoVisual(objetosNaEstrada);

		//se não exite onjetos no campo visual a normalmente
		if(objetosNoCampoVisual.size() == 0){//anda normalmente
```

```
				this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de
um em um segundo, supondo a velocidade em m/s
			}
			else{
				//verifica quais objetos estão na frente
				for(int i=0; i<objetosNoCampoVisual.size(); i++){
					Object obj = objetosNoCampoVisual.get(i);

					//se um semaforo
					if(obj instanceof Semaforo){

						Semaforo sem = (Semaforo) obj;

	if(this.getNomeDaEstrada().equals(sem.getNomeDaEstrada1()) && this.getVelocidade()
!= -1){

							if(sem.getCor_1().equals("verde") ||
sem.getCor_1().equals("amarelo")){
									if(this.getVelocidade() == 0){

	this.setVelocidade(this.getVelocidadeInicial());
									}

	this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um
segundo, supondo a velocidade em m/s
							}
							else{
								if(this.getVelocidade() == 0){

	System.out.println("[velocidade = 0] Veículo parado");

	TesteDaSimulacao.tempoNaFilaEstrada1++;

								}
								else{

									float proxima_posicao =
this.getPosicao_atual()+this.getVelocidade()*1;

									//verifica se o que deve andar é
maior que a distancia do objeto
									if(proxima_posicao >
sem.getPosicao_1()){//se for -> para a um metro do objeto
										proxima_posicao =
sem.getPosicao_1() - 1;//considerando que estamos lidando com distancia em metro

	setPosicao_atual(proxima_posicao);

	this.setVelocidade(0);//para o carro
```

```
                                    }
                                    else{//se não for -> andar
normal

        this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um
segundo, supondo a velocidade em m/s
                                    }
                                    //debug
                                    System.out.println("[id]: " +
this.getIdCarro());
                                }

                            }
                    }
                    else
if(this.getNomeDaEstrada().equals(sem.getNomeDaEstrada2()) && this.getVelocidade() != -1){

                        if(sem.getCor_2().equals("verde") ||
sem.getCor_2().equals("amarelo")){
                            if(this.getVelocidade() == 0)

        this.setVelocidade(this.getVelocidadeInicial());

        this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um
segundo, supondo a velocidade em m/s
                        }
                        else{
                            if(this.getVelocidade() == 0){
                                System.out.println("[velocidade = 0]
Veículo parado");

        TesteDaSimulacao.tempoNaFilaEstrada2++;
                            }
                            else{
                                float proxima_posicao =
this.getPosicao_atual()+this.getVelocidade()*1;
                                //verifica se o que deve andar é maior
que a distancia do objeto
                                if(proxima_posicao >=
sem.getPosicao_2()){//se for -> para a um metro do objeto
                                    proxima_posicao =
sem.getPosicao_2() - 1;//considerando que estamos lidando com distancia em metro

        setPosicao_atual(proxima_posicao);
                                    this.setVelocidade(0);
                                }
                                else{//se não for -> andar normal
```

```
this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um segundo, supondo a velocidade em m/s
                                        }
                                        //debug
                                        System.out.println("[id]: " + this.getIdCarro());
                                    }

                                }
                            }
                            else if(this.getNomeDaEstrada().equals(sem.getNomeDaEstrada2()) && this.getVelocidade() != -1){
                                if(sem.getCor_2().equals("verde") || sem.getCor_2().equals("amarelo")){

                                    if(this.getVelocidade() == 0)

this.setVelocidade(this.getVelocidadeInicial());


this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um segundo, supondo a velocidade em m/s
                                }
                                else{
                                    if(this.getVelocidade() == 0){
                                        System.out.println("[velocidade = 0]");

TesteDaSimulacao.tempoNaFilaEstrada3++;
                                    }
                                    else{
                                        float proxima_posicao = this.getPosicao_atual()+this.getVelocidade()*1;

                                        //verifica se o que deve andar é maior que a distancia do objeto
                                        if(proxima_posicao > sem.getPosicao_2()){//se for -> para a um metro do objeto
                                            proxima_posicao = sem.getPosicao_2() - 1;//considerando que estamos lidando com distancia em metro

setPosicao_atual(proxima_posicao);
                                            this.setVelocidade(0);
                                        }
                                        else{//se não for -> andar normal

this.setPosicao_atual(this.getPosicao_atual()+this.getVelocidade()*1);//de um em um segundo, supondo a velocidade em m/s
```

```
                                                                        }
                                                                        //debug
                                                                        System.out.println("[id]: " +
this.getIdCarro());
                                                                }
                                                        }
                                                }

                                        }//fecha (obj...

                                }//fecha for(int i...

                        }

                }//fecha calculaProximaPosicao

                public int getIdCarro(){
                        return idCarro;
                }

                public void setIdCarro(int idCarro){
                        this.idCarro = idCarro;
                }

                public float getCampoVisual() {
                        return campoVisual;
                }

                public void setCampoVisual(float campoVisual) {
                        this.campoVisual = campoVisual;
                }

                /**
                 * Este método verifica se tem algo no campo visual do carro
                 */
                public ArrayList temAlgoNoCampoVisual(ArrayList objetosNaEstrada){

                        int j = 0;
                        Semaforo sem;
                        Carro carro;
                        ArrayList retorno = new ArrayList();//objetos que estão no campo visual

                        Iterator it = objetosNaEstrada.iterator();

                        for(int i=0; i<objetosNaEstrada.size(); i++){
                                Object obj = it.next();
                                if(obj instanceof Semaforo){//se o objeto é Um Semaforo
                                        sem = (Semaforo) obj;
                                        if(this.getNomeDaEstrada().equals("Estrada 1")){
```

```
                    if(sem.getPosicao_1()> this.getPosicao_atual() &&
sem.getPosicao_1() <= (this.getPosicao_atual()+this.getCampoVisual())){
                        retorno.add(j++, sem);
                    }
                }
                else if(this.getNomeDaEstrada().equals("Estrada 2")){
                    if(sem.getPosicao_2()> this.getPosicao_atual() &&
sem.getPosicao_2() <= (this.getPosicao_atual()+this.getCampoVisual())){
                        retorno.add(j++, sem);
                    }
                }
                else if(this.getNomeDaEstrada().equals("Estrada 3")){
                    if(sem.getPosicao_2()> this.getPosicao_atual() &&
sem.getPosicao_2() <= (this.getPosicao_atual()+this.getCampoVisual())){
                        retorno.add(j++, sem);
                    }
                }
            }//fecha if(obj...
            if(obj instanceof Carro){//se o objeto é um Carro
                carro = (Carro) obj;
                if(carro.getPosicao_atual()> this.getPosicao_atual() &&
carro.getPosicao_atual() < (this.getPosicao_atual()+this.getCampoVisual())){
                    retorno.add(j++, carro);
                }
            }//fecha if(obj...

        }//fecha for(int i...

        return retorno;
    }//fecha temAlgoNoCampoVisual

}//fecha Carro


/**
*Classe Estrada
*/

/*
 * Created on 14/03/2006
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package Basico;

import java.util.ArrayList;
import java.util.Iterator;
```

```
/**
 * @author Frédick
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Estrada {

	private float comprimento;
	private String nomeDaEstrada;
	public static boolean estradaCriada = false;
	private ArrayList objetosNaEstrada;

	public Estrada(String nome, float comprimento/*, ArrayList semaforos*/){

		setNomeDaEstrada(nome);
		setComprimento(comprimento);
		objetosNaEstrada  = new ArrayList();
		setEstradaCriada(true);
		//System.out.println("[EstradaCriada]: " + this.isEstradaCriada());//debug

	}//fecha Estrada

	public float getComprimento() {
		return comprimento;
	}

	public void setComprimento(float comprimento) {
		this.comprimento = comprimento;
	}

	public String getNomeDaEstrada(){
		return nomeDaEstrada;
	}

	public void setNomeDaEstrada(String nomeDaEstrada){
		this.nomeDaEstrada = nomeDaEstrada;
	}

	public boolean isEstradaCriada(){
		return estradaCriada;
	}

	private void setEstradaCriada(boolean estradaCriada){
		this.estradaCriada = estradaCriada;
	}

	public void adicionaObjetosNaEstrada(int id, Object objeto){
		objetosNaEstrada.add(id, objeto);
```

```
	}

	public void adicionar(Object obj){
		objetosNaEstrada.add(obj);
	}

	public void removerObjetosDaEstrada(Carro car){
		objetosNaEstrada.remove(car.getIdCarro());
	}

	//retorno objetos validos
	public ArrayList getObjetosNaEstrada(){
//		return retorno;
		return objetosNaEstrada;
	}

	public void atualizarListaDeObjetosDaEstrada(String nomeDaEstrada){
		Iterator it = this.getObjetosNaEstrada().iterator();
		while(it.hasNext()){
			Object obj = it.next();
			if(obj instanceof Carro){
				Carro carro = (Carro) obj;
				if(carro.getPosicao_atual() >= this.getComprimento()){
					if(nomeDaEstrada.equals("Estrada 1")){
						if(TesteDaSimulacao.totalDeCarrosEstrada1 > 0){
							TesteDaSimulacao.totalDeCarrosEstrada1--;
						}
					}
					else if(nomeDaEstrada.equals("Estrada 2")){
						if(TesteDaSimulacao.totalDeCarrosEstrada2 > 0){
							TesteDaSimulacao.totalDeCarrosEstrada2--;

						}
					}
					else if(nomeDaEstrada.equals("Estrada 3")){
						if(TesteDaSimulacao.totalDeCarrosEstrada3 > 0){
							TesteDaSimulacao.totalDeCarrosEstrada3--;
						}
					}
					removerObjetosDaEstrada(carro);
					TesteDaSimulacao.tempoNaSaida.put(new
Integer(carro.getIdCarro()), new Integer(TesteDaSimulacao.horas));
				}
			}
		}
	}

}//fecha estrada
```

```
/**
*Semaforo
*/

/*
 * Created on 14/03/2006
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package Basico;

/**
 * @author Frédick
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Semaforo {

	private String cor_1;
	private String cor_2;
	private int tempo_verde    = 240;//segundos
	private int tempo_amarelo  = 120;
	private int tempo_vermelho = 360;
	private float posicao_1;
	private float posicao_2;
	private int tempo_cor_1 = 0;
	private int tempo_cor_2 = 0;
	private int idSemaforo;
	private String nomeDaEstrada1;
	private String nomeDaEstrada2;

	public Semaforo(String cor_1, String cor_2, String nomeEstrada1, String nomeEstrada2,
float posicao_1, float posicao_2, int id){
		this.setCor_1(cor_1);
		this.setCor_2(cor_2);
		this.setNomeDaEstrada1(nomeEstrada1);
		this.setNomeDaEstrada2(nomeEstrada2);
		this.setPosicao_1(posicao_1);
		this.setPosicao_2(posicao_2);
		this.setIdSemaforo(id);

	}

	public int getTempo_amarelo() {
		return tempo_amarelo;
	}
```

```
public void setTempo_amarelo(int tempo_amarelo) {
        this.tempo_amarelo = tempo_amarelo;
}

public int getTempo_verde() {
        return tempo_verde;
}

public void setTempo_verde(int tempo_verde) {
        this.tempo_verde = tempo_verde;
}

public int getTempo_vermelho() {
        return tempo_vermelho;
}

public void setTempo_vermelho(int tempo_vermelho) {
        this.tempo_vermelho = tempo_vermelho;
}

public float getPosicao_1() {
        return posicao_1;
}

public void setPosicao_1(float posicao) {
        this.posicao_1 = posicao;
}

public float getPosicao_2() {
        return posicao_2;
}

public void setPosicao_2(float posicao) {
        this.posicao_2 = posicao;
}

public String getCor_1() {
        return cor_1;
}

public void setCor_1(String cor_1) {
        this.cor_1 = cor_1;
}

public String getCor_2() {
        return cor_2;
}

public void setCor_2(String cor_2) {
```

```
		this.cor_2 = cor_2;
	}

	public int getIdSemaforo() {
		return idSemaforo;
	}

	public void setIdSemaforo(int idSemaforo) {
		this.idSemaforo = idSemaforo;
	}

	public String getNomeDaEstrada1() {
		return nomeDaEstrada1;
	}

	public void setNomeDaEstrada1(String nomeDaEstrada2) {
		this.nomeDaEstrada1 = nomeDaEstrada2;
	}

	public String getNomeDaEstrada2() {
		return nomeDaEstrada2;
	}

	public void setNomeDaEstrada2(String nomeDaEstrada2) {
		this.nomeDaEstrada2 = nomeDaEstrada2;
	}


	public void atualizarTempo(){

		if(this.getCor_1().equals("verde")){
				if(tempo_cor_1 == tempo_verde){
					tempo_cor_1 = 0;
					this.setCor_1("amarelo");
				}
				else{
					tempo_cor_1 = tempo_cor_1 + 1;
				}
			}
			else if (this.getCor_1().equals("amarelo")){
				if(tempo_cor_1 == tempo_amarelo){
					tempo_cor_1 = 0;
					this.setCor_1("vermelho");
				}
				else{
					tempo_cor_1 = tempo_cor_1 + 1;
				}
			}
			else if(this.getCor_1().equals("vermelho")){
```

```
                if(tempo_cor_1 == tempo_vermelho){
                        tempo_cor_1 = 0;
                        this.setCor_1("verde");
                }
                else{
                        tempo_cor_1 = tempo_cor_1 + 1;
                }
        }

        if(this.getCor_2().equals("verde")){
                if(tempo_cor_2 == tempo_verde){
                        tempo_cor_2 = 0;
                        this.setCor_2("amarelo");
                }
                else{
                        tempo_cor_2 = tempo_cor_2 + 1;
                }
        }
        else if (this.getCor_2().equals("amarelo")){
                if(tempo_cor_2 == tempo_amarelo){
                        tempo_cor_2 = 0;
                        this.setCor_2("vermelho");
                }
                else{
                        tempo_cor_2 = tempo_cor_2 + 1;
                }
        }
        else if(this.getCor_2().equals("vermelho")){
                if(tempo_cor_2 == tempo_vermelho){
                        tempo_cor_2 = 0;
                        this.setCor_2("verde");
                }
                else{
                        tempo_cor_2 = tempo_cor_2 + 1;
                }
        }
    //debug
    System.out.println("[tempo_cor_1]: " + tempo_cor_1  );
    System.out.println("[tempo_cor_2]: " + tempo_cor_2  );

  }//fecha atualizaTempo


}//fecha Semáforo

/**
* Classe TesteDaSimulação
*/
```

```
/*
 * Created on 16/03/2006
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package Basico;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

/**
 * @author Frédick
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class TesteDaSimulacao {

	private static int id = 100;//inicio da identificação dos carros
	public static HashMap tempoNaEntrada = new HashMap();//tempo que o carro entra na
estrada
	public static HashMap tempoNaSaida   = new HashMap();//tempo que o carro sai da
estrada
	public static int tempoNaFilaEstrada1 = 0;//tempo de espera na fila dos semáforos da
Estrada 1
	public static int tempoNaFilaEstrada2 = 0;
	public static int tempoNaFilaEstrada3 = 0;
	public static int totalDeCarrosEstrada1 = 0;//total de carro percorrendo a Estrada 1
	public static int totalDeCarrosEstrada2 = 0;
	public static int totalDeCarrosEstrada3 = 0;
	public static int horas = 0; //tempo da simulação

	public synchronized ArrayList gerarVeículos(int hora, String nomeDaEstrada){

		System.out.println("gerarVeiculos");

		ArrayList carros = new ArrayList();
		int ritmoDeChegada = 0;

		if(nomeDaEstrada.equals("Estrada 1")){
			if((hora>=6 && hora < 8) || (hora>=12 && hora < 14) || (hora>=18 &&
			hora < 20)){
				 ritmoDeChegada = (int) (Math.random() * 5);//gera de (0;4) carros
			}
			else if((hora>=8 && hora < 12) || (hora>=14 && hora < 18) || (hora>=20
			&& hora < 24)){
				ritmoDeChegada = (int) (Math.random() * 4);//gera de (0;3) carros
```

```
            }
            else if(hora>=0 && hora < 6){
                  ritmoDeChegada = (int) (Math.random() * 3);//gera de (0;2) carros
            }
      }
      else if(nomeDaEstrada.equals("Estrada 2")){
            if((hora>=6 && hora < 8) || (hora>=12 && hora < 14) || (hora>=18 &&
            hora < 20)){
                   ritmoDeChegada = (int) (Math.random() * 4);//gera de (0;3) carros
            }
            else if((hora>=8 && hora < 12) || (hora>=14 && hora < 18) || (hora>=20
            && hora < 24)){
                  ritmoDeChegada = (int) (Math.random() * 3);//gera de (0;2) carros
            }
            else if(hora>=0 && hora < 6){
                  ritmoDeChegada = (int) (Math.random() * 2);//gera de (0;1) carros
            }
      }
      else if(nomeDaEstrada.equals("Estrada 3")){
            if((hora>=6 && hora < 7) || (hora>=12 && hora < 14) || (hora>=19 &&
            hora < 24)){
                   ritmoDeChegada = (int) (Math.random() * 4);//gera de (0;3) carros
            }
            else if((hora>=7 && hora < 12) || (hora>=14 && hora < 19)){
                  ritmoDeChegada = (int) (Math.random() * 3);//gera de (0;2) carros
            }
            else if(hora>=0 && hora < 6){
                  ritmoDeChegada = (int) (Math.random() * 2);//gera de (0;1) carros
            }
      }

      for(int x=0; x< ritmoDeChegada; x++){
            float vel = 5 + (int)Math.random()*15;//de 5 a 19 m/s
            int campoVisual = 5 + (int)Math.random()*3; //campo visual de 5 a 7 m
            carros.add(new Carro(vel, 0, id++, campoVisual, nomeDaEstrada));
      }

      return carros;

}


public static void main(String args[]){

      //inicia construtor
      TesteDaSimulacao teste = new TesteDaSimulacao();

      //cria as estradas
      Estrada estrada1 = new Estrada("Estrada 1",100);//tamanho da estrada
```

```
Estrada estrada2 = new Estrada("Estrada 2",500);//tamanho da estrada
Estrada estrada3 = new Estrada("Estrada 3",700);//tamanho da estrada

//cria semaforos e diz a que estrada eles pertencem
Semaforo semaforo1 = new Semaforo("verde", "vermelho",
estrada1.getNomeDaEstrada(), estrada2.getNomeDaEstrada(), 300, 200, 10);
Semaforo semaforo2 = new Semaforo("vermelho", "verde",
estrada1.getNomeDaEstrada(), estrada3.getNomeDaEstrada(), 700, 300, 11);


//adicionar semaforos na estrada
if(estrada1.isEstradaCriada()){
        estrada1.adicionar(semaforo1);
        estrada1.adicionar(semaforo2);
}

if(estrada2.isEstradaCriada()){
        estrada2.adicionar(semaforo1);
}

if(estrada3.isEstradaCriada()){
        estrada3.adicionar(semaforo2);
}

ArrayList objetos;

if(estrada1.isEstradaCriada()& estrada2.isEstradaCriada() &
estrada3.isEstradaCriada()){//verifica se estrada esta criada

        for(; horas < 24; horas++){//equivalente a 24 horas de simulação

                //quantidade de segundos em uma hora
                for(int segundos =1; segundos<=3600; segundos++ ){
                        System.out.println("*****HORAS: "+horas);//degug
                        System.out.println("*****SEGUNDOS: "+segundos);

                        //controla velocidade da simulação
                        for(int w=0; w<90000000; w++);


                        //HORARIOS DA ESTRADA 1
                        if((horas>=6 && horas<8) || (horas>=12 && horas<14) ||
                        (horas>=18 && horas<20)){//HOR´SRIO DE PIQUE
                                if(segundos%60 == 0){
                                        Iterator it = (teste.gerarVeículos(horas,
                                        estrada1.getNomeDaEstrada())).iterator();
                                        while(it.hasNext()){
                                                Carro carro = (Carro) it.next();
                                                estrada1.adicionar(carro);
```

```
                                    totalDeCarrosEstrada1++;
//                                  guarda o tempo que entrou no sistema
                                    tempoNaEntrada.put(new
                                    Integer(carro.getIdCarro()), new
                                    Integer(horas));
                              }

                        }
                  }
                  if((horas>=8 && horas<12) || (horas>=14 && horas<18) ||
                        (horas>=20 && horas<24)){//HORÁRIO NORMAL
                        if(segundos%120 == 0){
                              Iterator it = (teste.gerarVeículos(horas,
                              estrada1.getNomeDaEstrada())).iterator();
                              while(it.hasNext()){
                                    Carro carro = (Carro) it.next();
                                    estrada1.adicionar(carro);
                                    totalDeCarrosEstrada1++;
//                                  guarda o tempo que entrou no sistema
                                    tempoNaEntrada.put(new
                                    Integer(carro.getIdCarro()), new
                                    Integer(horas));
                              }

                        }
                  }
                  //HORÁRIO DE MADRUGADA
                  if((horas>=0 && horas<6)){

                        if(segundos%3600 == 0){
                              Iterator it = (teste.gerarVeículos(horas,
                              estrada1.getNomeDaEstrada())).iterator();
                              while(it.hasNext()){
                                    Carro carro = (Carro) it.next();
                                    estrada1.adicionar(carro);
                                    totalDeCarrosEstrada1++;
//                                  guarda o tempo que entrou no sistema
                                    tempoNaEntrada.put(new
                                    Integer(carro.getIdCarro()), new
                                    Integer(horas));
                              }

                        }
                  }

                  //HORÁRIOS DA ESTRADA 2
                  if((horas>=6 && horas<8) || (horas>=12 && horas<14) ||
                        (horas>=18 && horas<20)){
                        if(segundos%60==0){
```

```
                        Iterator it = (teste.gerarVeículos(horas,
                        estrada2.getNomeDaEstrada())).iterator();
                        while(it.hasNext()){
                                Carro carro = (Carro) it.next();
                                estrada2.adicionar(carro);
                                totalDeCarrosEstrada2++;
//                              guarda o tempo que entrou no sistema
                                tempoNaEntrada.put(new
                                Integer(carro.getIdCarro()), new
                                Integer(horas));
                        }

                }
        }
        if((horas>=8 && horas<12) || (horas>=14 && horas<18) ||
                (horas>=20 && horas<24)){
                if(segundos%120 == 0){
                        Iterator it = (teste.gerarVeículos(horas,
                        estrada2.getNomeDaEstrada())).iterator();
                        while(it.hasNext()){
                                Carro carro = (Carro) it.next();
                                estrada2.adicionar(carro);
                                totalDeCarrosEstrada2++;
//                              guarda o tempo que entrou no sistema
                                tempoNaEntrada.put(new
                                Integer(carro.getIdCarro()), new
                                Integer(horas));
                        }

                }
        }
        if((horas>=0 && horas<6)){
                if(segundos%3600==0){
                        Iterator it = (teste.gerarVeículos(horas,
                        estrada2.getNomeDaEstrada())).iterator();
                        while(it.hasNext()){
                                Carro carro = (Carro) it.next();
                                estrada2.adicionar(carro);
                                totalDeCarrosEstrada2++;
//                              guarda o tempo que entrou no sistema
                                tempoNaEntrada.put(new
                                Integer(carro.getIdCarro()), new
                                Integer(horas));
                        }

                }
        }

        //HORARIOS DA ESTRADA 3
```

```
if((horas>=6 && horas<8) || (horas>=12 && horas<14) ||
(horas>=18 && horas<20)){
	if(segundos%60==0){
		Iterator it = (teste.gerarVeículos(horas,
		estrada3.getNomeDaEstrada())).iterator();
		while(it.hasNext()){
			Carro carro = (Carro) it.next();
			estrada3.adicionar(carro);
			totalDeCarrosEstrada3++;
//			guarda o tempo que entrou no sistema
			tempoNaEntrada.put(new
			Integer(carro.getIdCarro()), new
			Integer(horas));
		}

	}
}
if((horas>=8 && horas<12) || (horas>=14 && horas<18) ||
(horas>=20 && horas<24)){
	if(segundos%120==0){
		Iterator it = (teste.gerarVeículos(horas,
		estrada3.getNomeDaEstrada())).iterator();
		while(it.hasNext()){
			Carro carro = (Carro) it.next();
			estrada3.adicionar(carro);
			tempoNaEntrada.put(new
			Integer(carro.getIdCarro()), new
			Integer(horas));
			totalDeCarrosEstrada3++;
//			guarda o tempo que entrou no sistema
			tempoNaEntrada.put(new
			Integer(carro.getIdCarro()), new
			Integer(horas));
		}

	}
}
if((horas>=0 && horas<6)){
	if(segundos%3600==0){
		Iterator it = (teste.gerarVeículos(horas,
		estrada3.getNomeDaEstrada())).iterator();
		while(it.hasNext()){
			Carro carro = (Carro) it.next();
			estrada3.adicionar(carro);
			totalDeCarrosEstrada3++;
//			guarda o tempo que entrou no sistema
			tempoNaEntrada.put(new
			Integer(carro.getIdCarro()), new
			Integer(horas));
```

```
                    }

            }
    }

    //Atualiza os objetos da Estrada 1
    objetos = estrada1.getObjetosNaEstrada();
    Iterator it = objetos.iterator();

    while(it.hasNext()){

            Object obj = it.next();

            if(obj instanceof Semaforo){
                    Semaforo sem = (Semaforo) obj;
                    sem.atualizarTempo();
            }
            else if (obj instanceof Carro){
                    Carro car = (Carro) obj;
                    //System.out.println("IdCarro: " +
car.getIdCarro());

        car.calculaProximaPosicao(estrada1.getObjetosNaEstrada());

            }

    }//fecha while(it...

    //Atualiza os objetos da Estrada 2
    objetos = estrada2.getObjetosNaEstrada();
    it = objetos.iterator();

    while(it.hasNext()){
            Object obj = it.next();

            if(obj instanceof Semaforo){
                    Semaforo sem = (Semaforo) obj;
                    sem.atualizarTempo();
            }
            else if (obj instanceof Carro){
                    Carro car = (Carro) obj;
                    //System.out.println("IdCarro: " +
car.getIdCarro());

        car.calculaProximaPosicao(estrada2.getObjetosNaEstrada());

            }

    }//fecha while(it...
```

```
                                        //Atualiza os objetos da Estrada 3
                                        objetos = estrada3.getObjetosNaEstrada();
                                        it = objetos.iterator();

                                        while(it.hasNext()){
                                              Object obj = it.next();

                                              if(obj instanceof Semaforo){
                                                    Semaforo sem = (Semaforo) obj;
                                                    sem.atualizarTempo();
                                              }
                                              else if (obj instanceof Carro){
                                                    Carro car = (Carro) obj;
                                                    //System.out.println("IdCarro: " +
car.getIdCarro());

      car.calculaProximaPosicao(estrada3.getObjetosNaEstrada());

                                              }

                                        }//fecha while(it...

                                        //atualiza as estradas 1, 2 e 3

      estrada1.atualizarListaDeObjetosDaEstrada(estrada1.getNomeDaEstrada());

      estrada2.atualizarListaDeObjetosDaEstrada(estrada2.getNomeDaEstrada());

      estrada3.atualizarListaDeObjetosDaEstrada(estrada3.getNomeDaEstrada());

                              }

                        }//fecha for(int i...
                  }

      }//fecha main

}//fecha TesteDaSimulacao
```

Anexo V

Tabelas de modelos simulados e de resultados da simulação

Aqui apresentamos a tabela que lista os modelos e seus parâmetros. Além da tabela de resultados nas duas simulações, primeiramente no Arena depois em Java.

Tabela 12 Lista dos Modelos simulados e seus parâmetros

Modelo	Estradas			Semáforo		
	Estrada 1	Estrada 2	Estrada 3	Verde	Amarelo	Vermelho
1	1000	500	700	4	2	6
2	1000	500	700	2	1	3
3	1000	500	350	4	2	6
4	1000	500	350	2	1	3
5	1000	250	700	4	2	6
6	1000	250	700	2	1	3
7	1000	250	350	4	2	6
8	1000	250	350	2	1	3
9	500	500	700	4	2	6
10	500	500	700	2	1	3
11	500	500	350	4	2	6
12	500	500	350	2	1	3
13	500	250	700	4	2	6
14	500	250	700	2	1	3
15	500	250	350	4	2	6
16	500	250	350	2	1	3

Tabela 13. Resultado completo dos testes no Arena.

Modelo	Tempo no sistema (minuto)	Tempo na fila (minuto)	Quantidade de entidades		
			Estrada 1	Estrada 2	Estrada 3
1	11	4	1074	1379	1259
2	8	4	1115	1403	1278
3	10	4	1074	1397	1259
4	7	4	1115	1411	1278
5	10	4	1074	1379	1280
6	8	5	1111	1405	1250
7	9	3	1074	1397	1280
8	7	3	1115	1411	1258
9	10	4	1022	1379	1259
10	7	4	1024	1403	1278
11	10	3	1022	1397	1259
12	6	3	1024	1411	1278
13	10	4	1022	1379	1280
14	7	3	1024	1403	1258
15	9	3	1022	1397	1280
16	6	3	1024	1411	1258

Tabela 14. Resultado completo dos testes em Java.

Modelo	Tempo no sistema (minuto)	Tempo na fila (minuto)	Quantidade de entidades		
			Estrada 1	Estrada 2	Estrada 3
1	13	6	324	509	639
2	9	4	375	528	663
3	12	5	314	499	637
4	10	3	365	518	651
5	10	4	334	574	694
6	7	4	371	491	665
7	8	4	304	510	627
8	7	3	355	508	661
9	12	4	1272	1629	1509
10	9	3	1314	1610	1490
11	12	5	1272	1622	1512
12	8	4	1255	1602	1497
13	12	4	1266	1646	1515
14	6	3	1285	1623	1498
15	10	4	1272	1633	1495
16	6	3	1305	1601	1495