Délcio de Jesus Machado

algSim – Linguagem algorítmica para simulação de redes de filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto
2008

Délcio de Jesus Machado

algSim – Linguagem algorítmica para simulação de redes de filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientadora:
Profa. Dra. Renata Spolon Lobato

São José do Rio Preto
2008

Délcio de Jesus Machado

algSim – Linguagem algorítmica para simulação de redes de filas

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

_____	_____
Profa. Dra. Renata Spolon Lobato	Délcio de Jesus Machado

Banca Examinadora:
Prof. Dr. Aleardo Manacero Júnior
Prof. Dr. Mário Luiz Tronco

São José do Rio Preto
2008

Aos meus queridos pais Teotônio e Bernardeth, à minha irmã Denise e ao meu irmão Éder.

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus por todas as oportunidades que me tem dado.

Aos meus pais, Tito e Deth, por tudo que fizeram e têm feito por mim, por todas as dificuldades que tiveram que contornar para que nada me faltasse. E acima de tudo por estarem sempre por perto em todos os momentos da minha vida, prontos para chorar ou rir.

Aos meus irmãos Éder e Denise que apesar da distância nunca deixaram de se fazer presentes em minha vida e de me dar força. Por terem sempre acreditado em mim e me apoiado.

À minha orientadora e amiga Profa. Dra. Renata Spolon Lobato, que nunca deixou de ajudar-me diante de situações difíceis.

A todos os professores e funcionários do IBILCE que direta ou indiretamente participaram e contribuíram para a minha formação.

A todos meus amigos e colegas que tive que deixar para trás, aos que fiz durante os meus anos de faculdade e aos que estiveram sempre comigo em todos os momentos bons e ruins.

"A linguagem é apenas o instrumento da ciência, e as palavras não passam de símbolos das idéias."

Samuel Johnson (1709 – 1784)

RESUMO

Atualmente existem disponíveis várias linguagens para simulação de sistemas. Porém, a necessidade de conhecimento das estruturas e detalhes da linguagem de simulação escolhida para a implementação dos modelos pode criar enormes dificuldades aos usuários dessas linguagens, devido ao tempo exigido durante o aprendizado das mesmas. Neste trabalho desenvolveu-se uma linguagem algorítmica (simples e em português estruturado) para simulação de redes de filas – denominada algSim - e desenvolveu-se também um compilador para a referida linguagem. Essa linguagem tem como objetivo auxiliar os programadores que possuem pouca familiaridade com as linguagens de simulação na construção de modelos de redes de filas para simulação de sistemas. Visando-se facilitar ainda mais as atividades do programador durante desenvolvimento de um programa na linguagem (etapas de edição, compilação e execução), criou-se também uma interface gráfica para algSim. O ambiente de programação implementado foi desenvolvido sob a plataforma GNU/Linux, com a distribuição Ubuntu. Para a concepção do ambiente algSim utilizaram-se as ferramentas *GNU Compiler Collection* (GCC) e *Java Development Kit (JDK).*

Abstract

Currently there are multiple simulating systems languages. However, the need for knowledge of the structures and details of the simulation language chosen for the implementation of the models creates enormous difficulties for users of these languages because of the time required for the learning of them. In this work was developed an algorithmic language (simple and in structured portuguese) for queuing networks simulation - algSim - and it was also developed a compiler for the referred language. The language developed aims to help the developers who have little familiarity with simulation languages to construct queuing networks models for simulation systems. Aiming to facilitate further the activities of the programmer during the development of a program using the language (stages of editing, compilation and execution), it was also created a graphical interface for algSim. The programming environment described here was developed under the GNU/Linux platform, with the Ubuntu distribution. For the design of the algSim environment it were used the tools GNU Compiler Collection (GCC) and Java Development Kit (JDK).

Índice

Lista de Figuras

Lista de Abreviaturas e Siglas

algSim: Linguagem algorítmica para a simulação de redes de filas
ACM: *Association for Computer Machinery*
AMD: *Advanced Micro Devices*
ASCII: *American Standard Code for Information Interchange*
BNF: *Backus-Naur Form*
CISC: *Complex Instruction Set Computer*
CPU: *Central Processing Unit*
DCCE: *Departamento de Ciência da Computação e Estatística*
ELF: *Executable and Linking Format*
FCFS: *First Come First Served*
FIFO: *First In First Out*
gcc: *GNU C Compiler*
GCC*: GNU Compiler Collection*
GNU: *GNU is Not Unix*
GSPD: *Grupo de Sistemas Paralelos e Distribuídos*
IA-32: *Intel 32-bit Architecture*
IA-64: *Intel 64-bit Architecture*
IDE: *Integrated Development Environment*
IBILCE: *Instituto de Letras e Ciências Exatas*
JDK: *Java Development Kit*
LEF: Lista de Eventos Futuros
LisRef: *Linguagem algorítmica para a simulação de redes de filas*
LIFO: *Last In First Out*
RFC: *Request For Comments*
RFOO: Redes de Filas Orientadas a Objetos
RR: *Round Robin*
UML: *Unified Modeling language*

Capítulo 1 – Introdução

1.1 Considerações Iniciais

Em geral, a concepção de sistemas de grande porte envolve uma grande quantidade de recursos, o que inclui o esforço necessário para o seu desenvolvimento e os custos necessários para a sua implementação. Em razão disso, torna-se imprescindível a obtenção de garantias de que o sistema projetado irá funcionar de maneira ideal.

Para a obtenção de tais garantias, pode-se recorrer ao uso das técnicas de modelagem (solução analítica e simulação), que são geralmente utilizadas quando o sistema a ser modelado ainda é inexistente, sendo que nesse caso é necessária a concepção de um modelo do sistema a ser avaliado (Balieiro, 2005). A solução analítica envolve a modelagem dos sistemas em termos de equações e solução dos mesmos usando métodos matemáticos. Entretanto, muitos sistemas do mundo real são bastante complexos, o que faz com que os modelos desses sistemas sejam praticamente impossíveis de serem solucionados matematicamente (Banks *et al.*, 1996). Assim, como uma alternativa barata e confiável em relação à construção de um sistema real, surge a simulação que consiste na imitação da operação de um sistema ou processo do mundo real ao longo do tempo (Banks *et al.*, 1996).

Uma das principais vantagens na realização da simulação é a não necessidade de um sistema pronto, o que a torna adequada para o estudo de sistemas na fase de desenvolvimento ou que não podem ser testados (Banks *et al.*, 1996).

Contudo, existem várias situações, envolvendo principalmente usuários que

possuem poucos conhecimentos de simulação, em que se torna bastante complexo desenvolver um modelo que represente com fidelidade o cenário real. Porém, quando tal modelo for concebido de forma correta, a simulação torna-se uma ferramenta bastante confiável e eficaz.

Uma vez concebido de forma correta o modelo a ser simulado, o mesmo pode ser implementado utilizando linguagens de programação convencionais, extensões funcionais, pacotes de uso específico e linguagens de simulação (Santana *et al*, 1994).

Atualmente existem disponíveis várias linguagens para simulação de sistemas, dentre elas podemos citar a **GPSS/H, SimPy, ModSimIII, Simscript II.5, SmallDEVS**, etc (Banks *et al.*, 1996). Para que os modelos nelas implementados imitem as respostas dos sistemas reais em análise, utilizam-se conceitos de redes de filas que incluem entidades denominadas usuários e centros de serviços que são formados por servidores e filas. O único problema ao se trabalhar com esse tipo de linguagem (linguagens de simulação), é que elas exigem dos usuários (além do conhecimento de teoria de filas) conhecimentos das estruturas e detalhes da linguagem de simulação escolhida para a implementação do modelo, o que pode levar tempo para o aprendizado.

Como solução para as dificuldades encontradas pelos usuários no cenário mencionado anteriormente, projetou-se uma linguagem algorítmica simples e compacta em português estruturado para simulação de redes de filas, porém com todos os aspectos e estruturas de linguagem de simulação, necessárias para que os usuários possam criar modelos de simulação sem conhecimentos avançados sobre ferramentas concebidas para tal propósito.

1.2 Objetivos

Este projeto tem como objetivo a construção de uma versão de uma linguagem algorítmica para simulação de redes de filas – algSim. O projeto tem como base um trabalho desenvolvido no Grupo de Sistemas Paralelos e Distribuídos (GSPD) (Oliveira, 2006). No atual projeto implementa-se a fase de síntese e reestrutura-se a fase de análise do projeto anterior (LisRef), o que deu origem a uma

nova linguagem (algSim).

A melhoria adicionada por algSim em relação a LisRef é uma maior simplicidade na construção de algoritmos para a simulação de sistemas, permitindo assim aos usuários da linguagem simular modelos **M/M/K** (onde **M** representa as distribuições de chegada e serviço e **K** representa o número de servidores paralelos) através da definição dos seguintes elementos: modelo de simulação, recurso compartilhado, tempo entre chegadas, tempo de serviço, tempo de simulação, eventos do sistema, quantidade de filas de espera e a política de escalonamento.

A idéia é proporcionar ao usuário um nível de abstração maior no desenvolvimento dos seus modelos para simulação, através de uma linguagem em português estruturado de fácil entendimento e utilização, de modo que o usuário familiarizado com algoritmos e teoria de filas possa agilizar o processo de implementação/codificação do seu modelo.

A linguagem desenvolvida é modular, de modo a poder ser facilmente estendida no futuro para inclusão de mais recursos. Assim, nesta versão ela disponibiliza através da biblioteca "algSim.h" modelos básicos de redes filas - A/S/c/k/m, política de escalonamento do tipo *FIFO*, distribuições de probablidade exponenciais e uniformes, lista de eventos futuros, função aleatória, recursos, função para impressão de relatórios bem comandos e estruturas de dados necessárias que facilitam o desenvolvimento dos programas.

Para a linguagem algSim foi construído um compilador completo. As fases do *front end* e do *back end* foram implementadas usando a linguagem JAVA (Deitel & Deitel, 2004), sendo que o programa objeto foi gerado em linguagem ASSEMBLY (*inline assembly*) (Blum, 2005). Os módulos do *front end* e do *back end* foram construídos sob a plataforma *GNU/LINUX e* o código objeto gerado foi compilado e ligado com todos os outros arquivos objetos da biblioteca algSim.h através do compilador *GCC* (*GNU Compiler Collection*) (Grifith, 2002).

Foram fornecidos ao compilador vários programas fontes na linguagem algSim e analisada a sua saída, que resultava em erros casos eles existissem ou no programa objeto e a simulação do sistema desejado caso o programa fonte estivesse correto. A mensagem gerada pelo compilador, caso exista algum erro sintático, é o tipo de erro e a linha em que ocorreu. Para os outros tipos de erros (léxicos e semânticos) gerados nas fases de análise o compilador informa a linha em que o erro

ocorreu e o *token* esperado.

1.3 Organização da Monografia

No capítulo 2 desta monografia apresenta-se a revisão bibliográfica dos assuntos abordados neste projeto. No capítulo 3 apresenta-se a implementação da linguagem algSim e das bibliotecas que auxiliam a referida linguagem na construção dos algoritmos para simulação. No capítulo 4 são apresentados os resultados obtidos através dos testes realizados com a linguagem construída. Finalizando, apresentam-se no capítulo 5 as conclusões feitas sobre o trabalho realizado e sugestões de temas para trabalhos futuros.

Capítulo 2 – Revisão Bibliográfica

2.1 Considerações Iniciais

Neste capítulo abordam-se todos os conceitos teóricos fundamentais no desenvolvimento deste projeto. Inicialmente, faz-se um estudo sobre os conceitos básicos de simulação, conseqüentemente apresentam-se os conceitos teóricos sobre redes de filas, linguagens de programação e finalmente, discorre-se sobre os conceitos que envolvem a construção de compiladores.

2.2 Técnicas de Análise de Desempenho

Embora seja ignorada inúmeras vezes durante a concepção de um sistema, a análise de desempenho de sistemas computacionais exerce um papel de suma importância tanto em sistemas já existentes como em sistemas em desenvolvimento, uma vez que ela tem como objetivo determinar qual o dimensionamento mais adequado dos componentes do sistema de maneira que as metas pré-estabelecidas sejam alcançadas, o que traz ganhos imensuráveis em relação à má utilização dos recursos disponíveis e em relação ao tempo necessário para a obtenção dos resultados de determinados processamentos.

Dentre as técnicas de análise de desempenho existentes destacam-se as técnicas de aferição e as técnicas de modelagem. As técnicas de aferição (*benchmarking*, prototipação e coleta de dados) consistem em estudos realizados

sobre o próprio sistema em questão. Já as técnicas de modelagem (solução analítica e simulação) consistem na construção e análise de modelos representativos do sistema em questão.

As técnicas de aferição são adequadas para análise de desempenho de sistemas parcialmente construídos (prototipação) ou sistemas já totalmente construídos (*benchmarking* e coleta de dados). Por outro lado, as técnicas de modelagem são adequadas tanto para sistemas existentes (sistemas totalmente ou parcialmente disponíveis) ou inexistentes (sistemas em fase de estudo para construção).

Por apresentar maior flexibilidade em relação às técnicas de aferição, a solução de modelos através de simulação foi escolhida como base para o desenvolvimento desse projeto.

Na seção seguinte, revisam-se alguns conceitos relevantes das técnicas de modelagem.

2.2.1 Técnicas de Modelagem

Conforme mencionado na seção anterior, as técnicas de modelagem dividem-se em técnicas de solução analítica e técnicas de simulação.

- As técnicas de solução analítica consistem nos principais meios de predição de desempenho de sistemas que ainda não estão fisicamente disponíveis. Baseiam-se na construção de modelos matemáticos, como por exemplo, redes de Petri (Cardoso & Valette, 1996) e Cadeias de Markov. Quando a solução analítica do sistema existe, essas técnicas são consideradas de baixo custo. Porém, a criação de um modelo analítico preciso é muito difícil devido à complexidade no seu processo de desenvolvimento. Assim sendo, tais métodos têm aplicabilidade condicionada à obtenção de um equacionamento preciso e computacionalmente factível do sistema, no qual devem ser levados em consideração uma quantidade elevada de parâmetros (Marcari & Manacero, 2003).

- As técnicas de simulação são similares às técnicas de solução analíticas. A diferença entre ambas as técnicas reside na forma de obtenção de resultados. Nas técnicas de simulação, no lugar das equações matemáticas criam-se regras de comportamento que definem o comportamento dos eventos e dos estados do sistema (Marcari & Manacero, 2003). Grande parte dessas técnicas são baseadas na simulação de eventos em grafos dirigidos, redes de Petri e redes de filas. Neste projeto enfatiza-se abordagem de simulação utilizando redes de filas, a qual é revisada na seção 2.3.

Pelo fato das técnicas analíticas não serem capazes de resolver uma gama de problemas práticos em diversas áreas e considerando-se também que para efetuarem-se modificações no sistema modelado deve-se refazer todo modelo e ainda solucioná-lo (Santana *et al.,*1994), optou-se pela abordagem por técnicas de simulação já que elas permitem refletir essas alterações no modelo de maneira mais simplificada (Santana *et al.,*1994).

A subseção seguinte revisa alguns conceitos básicos relativos à simulação de sistemas, porém tais conceitos constituem uma das bases fundamentais para o desenvolvimento deste projeto.

2.2.2.1 Simulação

A simulação consiste na realização de experimentos sobre um modelo matemático ou lógico (que represente com fidelidade o sistema real) com o objetivo de prever o comportamento ou impacto de tal situação em sistema real.

Um dos aspectos importantes em simulação é caracterização dos tipos de modelos a serem simulados, os quais podem ser classificados em modelos discretos e contínuos.

Nos modelos discretos, as mudanças de estado ocorrem em pontos discretos do tempo, ao passo que nos modelos contínuos, essas mudanças ocorrem continuamente no tempo (Santana *et al.,*1994). Nesse projeto abordam-se os

conceitos relacionados a simulação por modelos de eventos discretos (Banks *et al*., 1996):

- **Sistema**: Coleção de entidades (como por exemplo, pessoas ou máquinas) que interagem entre si ao longo do tempo para a realização de um ou mais objetivos.
- **Modelo**: Representação abstrata de um sistema, normalmente contendo relações estruturais, lógicas ou matemáticas que descrevem um sistema em termos de estado, entidades e seus atributos, conjuntos, processos, eventos, atividades e atrasos.
- **Estado do sistema**: Coleção de variáveis que contêm toda informação necessária para descrever o sistema a qualquer instante.
- **Entidade**: Qualquer objeto ou componente no sistema que requer uma representação explícita no modelo. Como por exemplo, servidor, cliente, máquina, etc.
- **Atributos**: Propriedades de uma dada entidade. Como por exemplo, prioridade de uso, trajetos definidos, etc.
- **Lista**: Coleção de entidades permanentemente ou temporariamente associadas, ordenadas segundo um critério. Como por exemplo, *First Come, First Served* (FCFS) – primeiro a entrar, primeiro a ser servido, ou por prioridades.
- **Evento**: Uma ocorrência instantânea que muda o estado do sistema. Como por exemplo, a chegada de um novo cliente.
- **Notificação de evento**: Registro de um evento que deve ocorrer no instante atual ou em algum instante futuro, juntamente com quaisquer dados necessários a execução do evento. Esse registro deve incluir no mínimo o tipo de evento e o tempo do evento.
- **Lista de Eventos**: Também conhecida como lista de eventos futuros (LEF), é uma lista de notificação de eventos, ordenada pelo tempo de ocorrência do evento.
- **Atraso**: Duração de um tempo de tamanho inespecificado, que não se sabe até que termine. Como por exemplo, o atraso de um cliente em uma fila (FIFO) depende de como ocorrem os eventos anteriores a ele.
- **Relógio**: Variável que representa o tempo simulado, servindo para

controle do tempo virtual do sistema. Esse tempo não representa o tempo real do sistema.

A seguir apresenta-se uma breve descrição das possíveis formas de implementação de um modelo de simulação (Santana *et al.,*1994), conforme mencionado na seção 1.1:

- **Linguagens de programação convencionais**: Oferecem a vantagem ao programador (ou modelador) de ter a liberdade de escolher uma linguagem de programação que seja de seu conhecimento (como por exemplo, C, Pascal), porém o programador tem que desenvolver todas as estruturas exigidas num ambiente de simulação.
- **Extensões Funcionais**: Consistem no desenvolvimento de bibliotecas para simulação utilizando uma linguagem de programação de uso geral, como por exemplo, a linguagem C. Utilizando-se este enfoque tem-se os aspectos gerais de uma linguagem de programação unidos ao caráter específico de uma linguagem de simulação.
- **Pacotes de uso específico**: Nesse tipo de abordagem deve existir um pacote que se ajuste às necessidades do sistema sendo modelado. Essa alternativa representa para o programador (ou modelador) a aprendizagem de uma nova linguagem, além de fornecer pouca flexibilidade a mudanças, por ser muito específico para um determinado tipo de sistema.
- **Linguagens de simulação**: Nesse caso, as linguagens já contêm todas as estruturas necessárias para a criação de um ambiente de simulação, o que evita com o que o programador tenha que executar tal tarefa. Por serem linguagens menos específicas, permitem a simulação de um número maior de sistemas. A única desvantagem é que o usuário pode ter que aprender uma nova linguagem.

As linguagens de simulação classificam-se em orientadas a eventos e orientadas a processos. Nas linguagens orientadas a eventos, o modelador deve determinar os eventos que podem causar a mudança no estado do sistema e desenvolver um algoritmo associado a cada evento. A seqüência de execução desses eventos é determinada por uma estrutura denominada lista de eventos futuros, que

contém todos os eventos que ocorrerão durante a simulação (Santana *et al.,*1994). Nas linguagens orientadas a processos o programa é organizado como um conjunto de processos (descritos como procedimentos de uma linguagem de programação), os quais executam concorrentemente durante a simulação. Como na orientação a eventos, também existe uma estrutura de lista, com a diferença de que cada entrada na lista define um processo e seu ponto de reativação.

Das possíveis implementações citadas anteriormente, a linguagem desenvolvida nesse projeto (algSim) encontra-se na categoria de linguagens de simulação orientadas a eventos. A motivação para escolha do desenvolvimento de uma linguagem desse gênero, é que as mesmas tendem a impor uma visão global e de alto nível do sistema ao modelador e são as linguagens mais adequada para modelagem de sistemas de médio e pequeno porte (MacDougall, 1975), o que vai de encontro com os objetivos definidos para esse projeto.

Na próxima seção revisam-se alguns conceitos importantes sobre redes de filas.

2.3 Redes de Filas

Conforme visto na seção anterior, a simulação é um método que tenta imitar o comportamento do sistema em estudo (representado por um modelo). Esses modelos dos sistemas podem ser obtidos pela abstração de sua estrutura e características através de redes de filas (Santana *et al.,* 1994), concluí-se que a simulação de sistemas pode ser aplicada na análise dos modelos desenvolvidos através de redes de filas.

Uma rede de filas é constituída de entidades denominadas centros de serviço e conjunto de entidades denominadas usuários (ou clientes), os quais recebem serviços nos centros de serviços. Um centro de serviço é constituído de um ou mais servidores (representando recursos do sistema a ser modelado) que prestam serviço aos usuários, e uma área de espera (denominada fila) para usuários cujos pedidos excedem a capacidade do centro de serviço, e estão esperando pelo serviço requisitado (Santana *et al.,*1994).

A seguir apresentam-se alguns exemplos de cenários reais onde aplica-se a

teoria de filas e ilustram-se os modelos básicos de redes filas para a representação de cada um dos exemplos:

Toma-se como exemplo para a figura 2.1 um cinema com somente uma bilheteria, onde as pessoas se enfileiram para a compra de ingresso (Santana *et al.,*1994).

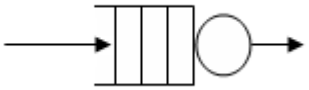


Figura 2.1 Centro de serviço: Uma fila e um servidor.

Um exemplo de um cenário real para figura 2.2 seria uma agência bancária, onde os caixas atendem aos clientes que formam uma única fila (Santana *et al.,*1994).

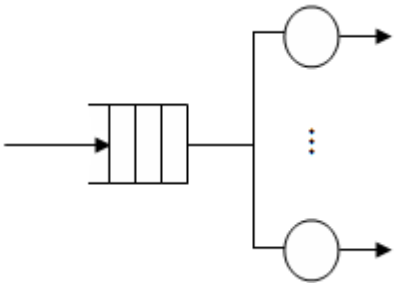


Figura 2.2 Centro de serviço: Uma fila e vários servidores.

Para a figura 2.3, podemos citar como exemplo um guichê de uma repartição pública que atende usuários que formam duas filas diferentes. Uma das filas se destina aos aposentados, e outra atende as demais pessoas. O atendente dá preferência à fila especial, só atendendo a outra fila quando não houver mais nenhum aposentado aguardando atendimento (Santana *et al.,*1994).

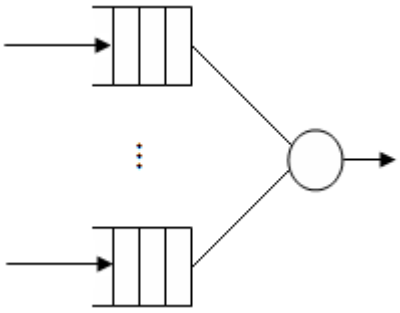


Figura 2.3 Centro de serviço: Múltiplas filas e um servidor.

Um exemplo para a figura 2.4, seria uma agência bancária onde os vários caixas atendem a clientes que formam duas filas: Uma para os clientes em geral e outra para idosos, gestantes e portadores de deficiências físicas. Os clientes em geral só são atendidos quando não houver ninguém na fila especial (Santana *et al.,*1994).

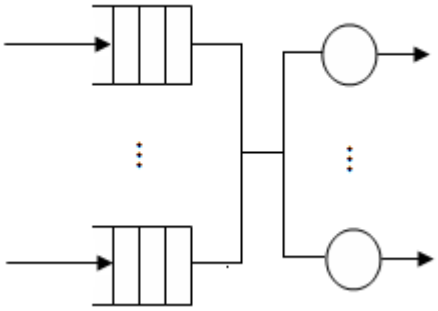


Figura 2.4 Centro de Serviço: Múltiplas filas e múltiplos servidores.

Para uma breve descrição do funcionamento de uma rede de filas aproveita-se o modelo ilustrado na figura 2.1, que é o modelo básico de redes filas. Nesse modelo os clientes chegam a algum servidor, e se o servidor estiver ocupado, o cliente une-se à fila associada a aquele servidor e aguarda a sua vez. Eventualmente, o cliente é selecionado para atendimento, de acordo com a disciplina da Fila. O serviço requisitado é então executado pelo servidor, e ao seu término o cliente abandona o sistema.

A seguir realçam-se alguns parâmetros importantes que devem ser levados em consideração na construção de modelos de simulação utilizando-se redes de filas:

- **Tempo entre-chegadas e Tempo de serviço**: O primeiro define a freqüência regular com que os clientes chegam ao sistema (Balieiro, 2005). O segundo define o tempo necessário para que o servidor execute uma tarefa. Por questões de simplicidade adota-se uma função

que produza valores constantes, porém tal adoção não permite representar o modelo de forma realista. Assim, é bastante comum para ambos, a adoção de funções que produzem valores aleatórios, atendendo a uma determinada distribuição de probabilidade, como por exemplo, distribuições exponenciais ou Markov (M), contantes ou determinísticas (D), Hiperexponenciais (H), k-Erlang (E_k), etc (Santana *et al.,*1994)(Banks *et al.*, 1996).

- **Disciplinas de Escalonamento**: A disputa por recursos do sistema geralmente leva a formação de linhas de espera (filas), onde os clientes aguardam sua vez de serem atendidos. Assim, a disciplina de fila é a regra (algoritmo) utilizada para adicionar e remover os clientes de uma fila. Dentre as regras que podem ser utilizadas destacam-se *FIFO (First In, First Out)* em que a ordem de atendimento obedece a ordem de chegada, *LIFO (Last In, First Out)* semelhante a uma pilha onde o último a entrar é o primeiro a sair, *RR (Round Robin)* onde cada cliente é atendido durante um certo intervalo de tempo denominado *quantum* time, etc (Santana *et al.,*1994).
- **Número de servidores**: Nos centros de serviços podem existir mais de um servidor. Nessa situação, quando um cliente requisita serviço, deve-se escolher qual dos servidores atenderá a essa requisição. O critério de escolha a ser utilizado, pode ser baseado no servidor que está a mais ou menos tempo desocupado, segundo algum tipo de prioridade ou aleatoriamente (Balieiro, 2005).
- **Capacidade do sistema**: Como nem todos os sistemas têm capacidade infinita em suas filas, quando o número de clientes excede a capacidade suportada, alguns clientes poderão perdidos ou descartados (Balieiro, 2005).

Atendendo a diversidade de sistemas de filas, conforme ilustrado nos exemplos desta seção, em 1953, *Kendall* propôs uma notação para a representação de sistemas com servidores paralelos que tem sido vastamente adotada. Essa notação tem forma A/B/c/N/K, na qual a letra A representa a distribuição do tempo entre chegadas, B diz respeito à distribuição de tempos de serviço, c representa o número

de servidores paralelos, N corresponde à capacidade do sistema e K representa o número de clientes na fonte geradora (Banks *et al.*, 1996).

Para além das áreas citadas nos exemplos ilustrados nesta seção, a teoria de filas pode ser empregue em muitas outras áreas, como por exemplo, sistemas de produção, sistemas de computador e comunicações, sistemas manuseamento de material e transporte, etc (Banks *et al.*, 1996).

2.4 Linguagem de Programação

Acredita-se que a profundidade de nossa capacidade intelectual seja influenciada pelo poder expressivo da linguagem em que comunicamos nossos pensamentos. Os que possuem uma compreensão limitada da linguagem natural têm enormes dificuldades diante da complexidade de expressar seus pensamentos, especialmente em termos de abstração de profundidade de abstração. Colocado de uma maneira simples, é difícil para pessoas descreverem estruturas que não podem descrever verbalmente ou por escrito (Sebesta, 2003).

Uma linguagem de programação pode ser definida como um método padronizado de expressar idéias para um computador.

As linguagens de programação permitem aos usuários (programadores) especificarem de maneira precisa, sobre quais dados um computador irá atuar, de que maneira esses dados devem ser transmitidos ou armazenados e sob quais circunstancias uma determinada ação deve ser tomada.

As linguagens de programação existentes podem ser classificadas segundo vários critérios como, paradigma, geração, estrutura de tipos e grau de abstração. Neste projeto da-se um enfoque especial à classificação das linguagens de programação segundo o grau de abstração.

De acordo com a *Association for Computer Machinery* (ACM), quanto ao grau de abstração as linguagens de programação classificam-se em linguagens de baixo, médio e alto nível.

As linguagens de baixo nível (ex: *ASSEMBLY*), estão diretamente ligadas às características e a arquitetura do computador, ou seja, são linguagens nas quais, o programador precisa ter conhecimentos das instruções e registradores do

processador.

As linguagens de alto nível (ex: JAVA), são as linguagens mais próximas a linguagem natural em relação as linguagens de baixo nível, ou seja, possuem um nível de abstração bastante elevado. O programador não precisa se preocupar com as instruções e registradores do processador.

As linguagens de médio nível (embora seja um termo que não seja aceite por muitos) são o meio termo entre as linguagens de alto e baixo nível. Como exemplo, podemos citar a linguagem C, uma vez que ela possui recursos de uma linguagem de alto nível, e ao mesmo tempo permite a manipulação direta de registradores, endereços de memórias, etc.

Conforme mencionado na seção 1.2, o objetivo deste projeto é elevar o nível de abstração na construção de programas de simulação, razão pela qual, optou-se pelo desenvolvimento de uma linguagem alto nível.

A subseção seguinte fornece uma visão geral sobre linguagens algorítmicas.

2.4.1 Linguagens Algorítmicas

Determinadas pessoas possuem enormes dificuldades para escrever a solução de um problema em uma linguagem de programação. Porém, a grande maioria das mesmas se sente muito à vontade na construção dos seus programas em algoritmos (Sebesta, 2003). Do exposto, fica visível que as pessoas em questão preferem descrever os passos para a solução de um problema (algoritmo) ao invés de descrever a seqüência das instruções do computador que devem ser executadas para a solução do problema (linguagem de programação) (Forbellone, 1999).

Tendo em conta o cenário anteriormente descrito, podemos definir uma linguagem algorítmica como sendo uma linguagem que associa o poder expressivo de uma linguagem de programação ao poder descritivo de um algoritmo na solução de um problema.

Geralmente, as linguagens algorítmicas são voltadas para o ensino de lógica de programação, porém a linguagem algSim é voltada para a construção de programas de simulação por parte de usuários que têm domínio de simulação, lógica e algoritmos mas que têm pouca afinidade com as linguagens de simulação

tradicionais.

Pela importância que os algoritmos possuem na construção da linguagem desenvolvida neste projeto e na construção de programas de computador em geral, na seção seguinte apresenta-se uma breve revisão dos mesmos.

2.4.2 Algoritmos

Um algoritmo é uma seqüência finita de passos que visa atingir um objetivo bem definido (Forbellone, 1999).

Os tipos de algoritmos mais utilizados são: descrição narrativa, fluxograma, portugol ou pseudocódigo. Embora utilizem metodologias diferentes, todos esses três tipos de algoritmos consistem em analisar o enunciado do problema e escrever os passos a serem executados para a solução do mesmo. Na descrição narrativa, a solução é escrita em uma linguagem natural (ex: português), o que pode levar a ambigüidade na interpretação da solução. No fluxograma, a solução é descrita através de símbolos gráficos, o que exige conhecimentos da notação gráfica utilizada, além de que essa descrição não apresenta muitos detalhes. No pseudocódigo, descreve-se a solução por intermédio de regras predefinidas, o que exige um aprendizado das mesmas (Ascencio & Campos, 2002).

Entretanto, por ter uma sintaxe simples, o pseudocódigo é a técnica de algoritmos adotada para a descrição de modelos de simulação na linguagem desenvolvida neste projeto (algSim). Enfatiza-se que tal escolha deveu-se ao fato de que, a sintaxe de uma linguagem de programação tem um impacto considerável na facilidade de uso da linguagem por parte dos programadores (usuários) e na legibilidade dos programas escritos na referida linguagem (Ghezzi & Jazayeri, 1991).

A seção seguinte aborda os possíveis métodos para a implementação de uma linguagem de programação.

2.5 Métodos de Implementação de uma Linguagem de Programação

Os principais métodos usados para a implementação de uma linguagem de programação são a compilação, a interpretação e a implementação híbrida (Sebesta, 2003).

Na compilação o programa fonte (linguagem-fonte) é traduzido para um programa equivalente em uma outra linguagem (linguagem alvo) (Aho *et al.,* 2007). A vantagem da compilação é o menor tamanho dos programas e a rapidez da execução do programa, assim que o processo de tradução for concluído (Sebesta, 2003). Porém tem como desvantagem a falta de portabilidade, uma vez que o código gerado é específico para uma determina arquitetura.

Na interpretação, ao invés de se produzir um programa alvo, executam-se diretamente as operações especificadas no programa fonte (Aho *et al.,* 2007). Tem como vantagem o aumento da portabilidade dos programas, porém os programas ocupam mais espaços de memória e a execução dos mesmos é de 10 a 100 vezes mais lenta do que a de programas compilados (Sebesta, 2003).

A implementação híbrida é um meio-termo entre a compilação e a interpretação. Nesse tipo de implementação o programa fonte é traduzido para uma linguagem intermediária projetada para permitir fácil interpretação (Sebesta, 2003)(Aho *et al.,* 2007). Se comparada com a interpretação, oferece a vantagem de maior rapidez na execução do programas. Comparada com a compilação é ligeiramente mais lenta, por acrescentar mais uma fase (interpretação).

Diante do exposto anteriormente, neste projeto optou-se pela construção de um compilador para a linguagem desenvolvida. Na seção seguinte faz-se uma breve revisão dos conceitos básicos envolvidos na construção de um compilador.

2.5.1Compilação

O processo de compilação desenvolve-se em diversas etapas, sendo que as mais importantes são ilustradas na figura 2.1.

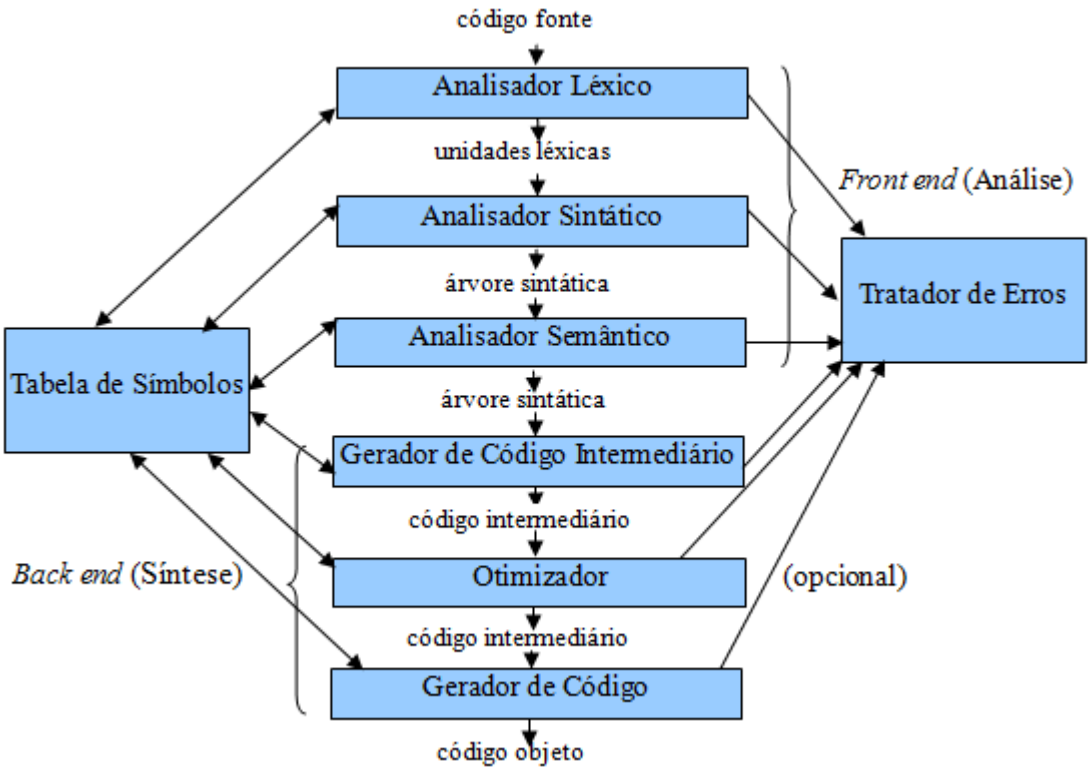


Figura 2.5 – Estrutura básica de um compilador (Aho *et al.,* 1995).

O analisador léxico, além de ignorar os comentários, reúne todos os caracteres do programa-fonte (código fonte) em unidades léxicas que são os identificadores, as palavras especiais, os operadores e símbolos de pontuação (Sebesta, 2003).

O analisador sintático tem como entrada as unidades do analisador léxico e usa as mesmas para a construção de estruturas hierárquicas denominadas árvores de análise, que representam a estrutura sintática do programa (Sebesta, 2003).

O analisador semântico verifica existem erros difíceis de serem detectados durante a análise sintática, como por exemplo, erros de tipos. Em muitas linguagens é responsável pela coerção de tipos.

O gerador de código intermediário resolve gradualmente a passagem do

código fonte para o código objeto. Esse fase oferece a vantagem de possibilitar a otimização do código.

O otimizador é uma parte opcional na compilação, e tenta melhorar o código intermediário de modo que o programa-alvo possa ser mais rápido em tempo de execução e/ou de menor tamanho.

O gerador de código é o responsável pela produção do código objeto, seleção de registradores, reserva de memória para variáveis, etc. Tem como entrada a representação intermediária do programa fonte e mapeia-o para a linguagem-alvo. Normalmente o código gerado está em linguagem de máquina relocável, linguagem absoluta de máquina ou em linguagem de montagem (Aho *et al.*, 1995).

A tabela de símbolos serve como banco de dados para o processo de compilação. Contém informações sobre os tipos e atributos de cada nome definido pelo usuário. Essas informações são colocadas na tabela pelos analisadores léxicos e sintáticos e usadas pelo analisador semântico e gerador de código.

Conforme ilustrado na figura 2.1, as fases de análise constituem o *front end* e as fases de geração de código constituem o *back end* do compilador.

O tratador de erros é usado pelo *front end* quando é detectada alguma incorreção léxica, sintática ou semântica no programa fonte. Para além da emissão de erros, o tratador deve emitir mensagens informativas, de modo que o usuário possa efetuar a devida correção e prosseguir com o processo de compilação.

Muitas vezes, além dos programas de sistemas, os programas de usuário devem ser vinculados a outros programas compilados anteriormente, que residem em bibliotecas. Para isso usa-se um linkeditor, que tem a função de vincular algum dado do programa aos programas do sistema e a outros programas do usuário (Sebesta, 2003).

2.6 Considerações Finais

Com a fundamentação teórica do projeto realizada, na qual abordaram-se aspectos como a análise de desempenho (com enfoque especial para simulação), modelagem de sistemas (onde enfatizou-se a implementação com redes de filas), linguagens de programação (na qual focaram-se especialmente as linguagens

algorítmicas, algoritmos) e os métodos para implementação de tais linguagens (com destaque para a compilação), no capítulo seguinte faz-se a descrição da implementação da linguagem e da interface gráfica para a edição de programas para a mesma.

Capítulo 3 – Desenvolvimento do Projeto

3.1 Considerações Iniciais

Neste capítulo apresenta-se todo o processo de desenvolvimento da interface gráfica (editor de código), da biblioteca e do compilador construído para a linguagem algSim. A especificação detalhada da construção da linguagem, como por exemplo, seus tipos de dados (numéricos, literais), palavras reservadas, comandos e as estruturas de dados auxiliares usadas, é feita durante o desenrolar do capítulo.

3.2 Estrutura do Projeto Desenvolvido

Nesta seção apresenta-se um resumo da descrição de funcionamento do projeto desenvolvido, conforme ilustrado no diagrama de atividades da UML (*Unified Modeling Language*) (Larman, 2004) mostrado na figura 3.1.

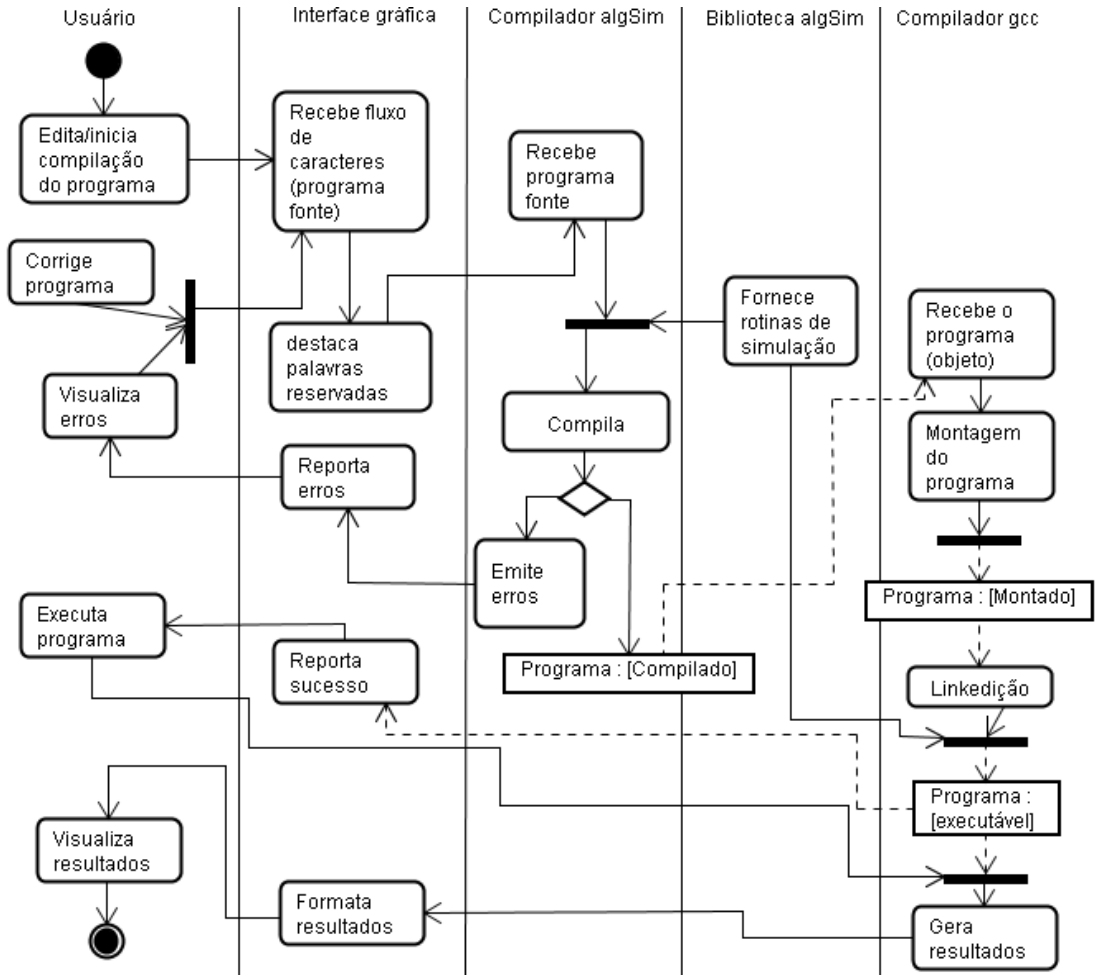


Figura 3.1 Diagrama de atividades do sistema.

O editor algSim (interface gráfica) tem como entrada o programa em algSim, e tem como saída uma seqüência de caracteres, que são passados ao compilador algSim (especificamente para o analisador léxico do compilador). No caso de presença de erros no arquivo passado ao compilador, os mesmos são mostrados na área de impressão de resultados da interface.

O compilador algSim produz o arquivo objeto (programa em *assembly*) para a arquitetura x86.

Terminada a fase de geração de código, dá-se início à fase de montagem do código objeto produzido pelo compilador e a ligação do mesmo com as rotinas de simulação necessárias presentes na biblioteca “algSim.h”. Esse processo é realizado através do compilador gcc, mediante o uso das ferramentas de montagem (*as* – GNU *Assembler*) e ligação (*ld* – GNU *linker*) que o mesmo possui. A saída do compilador gcc é um arquivo executável (arquivo em linguagem de máquina), pronto a ser

carregado para execução na memória.

O usuário executa o arquivo em linguagem de máquina através do uso da interface, mas pode fazê-lo também em linha de comando. O resultado da execução de tal arquivo (simulação) é passado para a interface gráfica, que se encarrega de imprimir os dados estatísticos da simulação na área do editor reservada para impressão de dados de saída.

Na figura 3.2 ilustra-se um diagrama de casos de uso (Larman, 2004) que representa as possíveis operações realizadas através da interação do usuário e a aplicação desenvolvida.

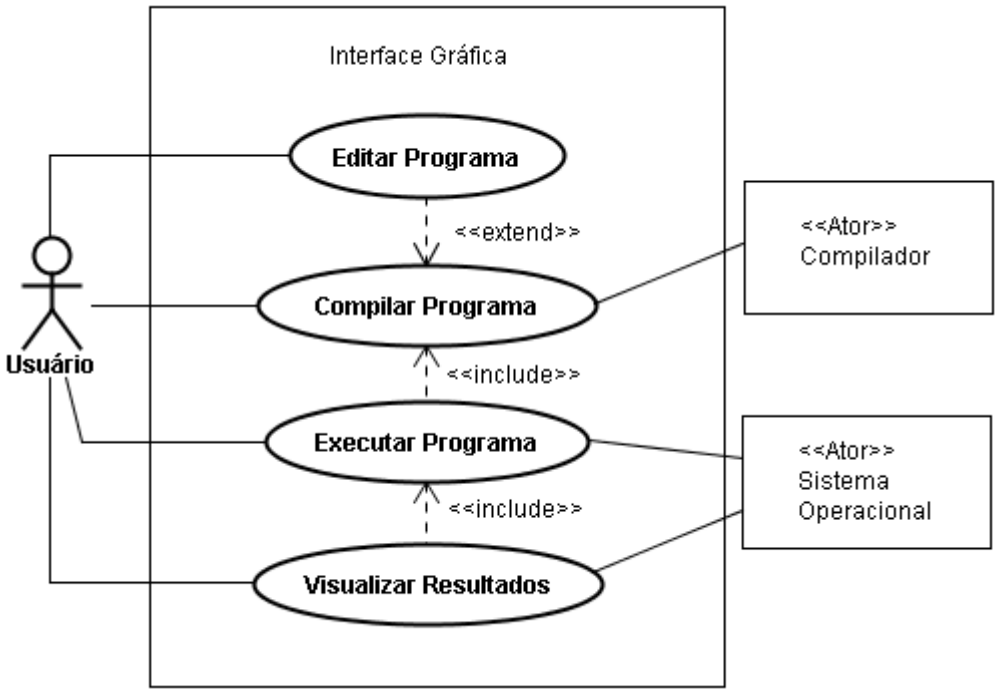


Figura 3.2 diagrama de casos de uso da interação entre o usuário e a aplicação.

3.3 Interface Gráfica

A implementação da interface gráfica foi feita em JAVA (Deitel & Deitel, 2004), através do ambiente de desenvolvimento integrado (IDE) Netbeans (Gonçalves, 2006) versão 6.0.1.

A interface gráfica auxilia o usuário na criação do código fonte através do reconhecimento de palavras da linguagem (*syntax highliting*), na

compilação/execução do programa que é feita de maneira automatizada através de um botão na barra de ferramentas (ou um menu para compilação/execução na barra de menus) ao invés de linha de comando (embora seja uma outra opção para o usuário) e na visualização dos resultados da simulação através da área de texto no editor reservada para a impressão de resultados (apesar de também poderem ser impressos diretamente no terminal de comandos).

Na figura 3.3 mostra-se em a interface gráfica, descreve-se alguns dos controles e o propósito de cada área dentro da mesma.

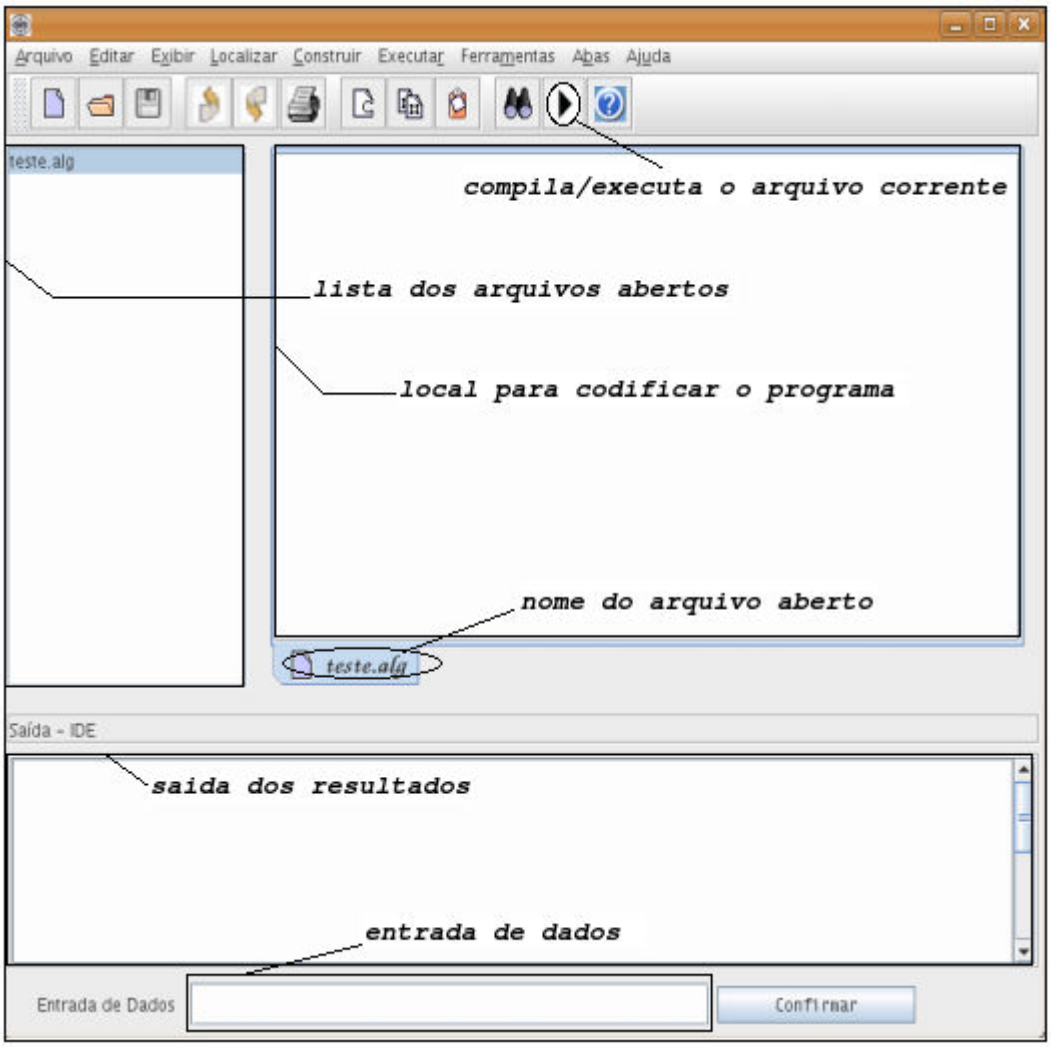


Figura 3.3 Interface gráfica.

3.4 Programa Fonte

A estrutura do programa fonte para a descrição dos modelos de sistema e simulação dos mesmos na linguagem algSim, é apresentada na figura 3.4.

```
/*
* Seção para declaração das bibliotecas
* usadas no programa
*/
      bibliotecas algSim
/*
* Seção para declaração do nome do algoritmo
*/
      algoritmo "Nome do algoritmo"
/*
* Seção para declaração das variáveis
* e execução dos comandos de simulação
*/
      /* Definição do modelo do sistema */
      modelo <identificador>{"distribuição de chegada",
                             "distribuição de serviço",
                             <numero_de_servidores>};
      /* Definição do recurso a ser compartilhado*/
      recurso <nome_do_recurso>;
      /* Definição do tempo entre chegadas*/
      defina tempo_entre_chegadas <tempo>;
      /* Definição do tempo de serviço*/
      defina tempo_de_servico <tempo>;
      /* Definição do tempo de simulação*/
      defina tempo_de_simulacao <tempo>;
      /* Definição dos eventos que ocorrerão
       * durante a simulação do sistema*/
      eventos <nome_do_recurso>{"evento de chegada",
                                "evento de atendimento",
                                "evento de saída"};
      /* Definiçao da fila associada ao recurso*/
      filas <nome_do_recurso> {"nome da fila"};
      /* Definição da disciplina de escalonamento */
      defina disciplina <nome_do_recurso><nº_de_recursos>
                             {<nome_da_disciplina>};
      /* Início da simulação*/
      simule <identificador>;
      /* Apresentação dos resultados */
      imprimeRelatorio();
      /* Fim do programa*/
      fimalgoritmo
```

Figura 3.4 Estrutura de um programa na linguagem algSim.

Na estrutura de código mostrada na figura 3.4, as palavras destacadas em azul e negrito (exceto imprimeRelatorio() - função), representam as palavras chaves da linguagem algSim. A inclusão da biblioteca algSim é opcional, uma vez que ela, quando não inclusa explicitamente no programa, é automaticamente inclusa pelo compilador algSim por ser a biblioteca padrão da linguagem. A especificação mais detalhada da linguagem algSim é apresentada na subseção 3.5.2 (analisador sintático).

3.5 Compilador da Linguagem

No desenvolvimento do compilador da linguagem, adotou-se a técnica de orientação a objeto ao invés da orientação a fase.

A vantagem da orientação a objeto em relação à orientação a fase é que na orientação a objeto, todo código referente a uma construção da linguagem está em uma classe, o que torna mais fácil a adição de novas construções, como por exemplo, laços (para, enquanto, etc.). Por outro lado, a vantagem da orientação a fase, é que torna mais fácil a adição ou modificação de uma fase, como por exemplo, verificação de tipos (análise semântica) (Aho *et al.,* 2007).

Levando-se em consideração a abordagem apresentada por ambos os métodos de desenvolvimento e os objetivos da linguagem algSim (elevar o grau de abstração na simulação do sistemas modelados), fica visível que futuramente, é muito mais provável que com o objetivo de simplificar ainda mais a linguagem aqui desenvolvida, se acrescentem a novas construções a linguagem em detrimento da adição de novas fases ao compilador (o que justifica a abordagem aqui escolhida).

Todos os módulos do compilador foram escritos em linguagem Java (JDK 6). O projeto foi desenvolvido sob a plataforma GNU/Linux (x86), kernel 2.6.24-19-Generic, distribuição Ubuntu-8.04 .

O diagrama de classes (Larman, 2004) do compilador desenvolvido é ilustrado na figura 3.5.

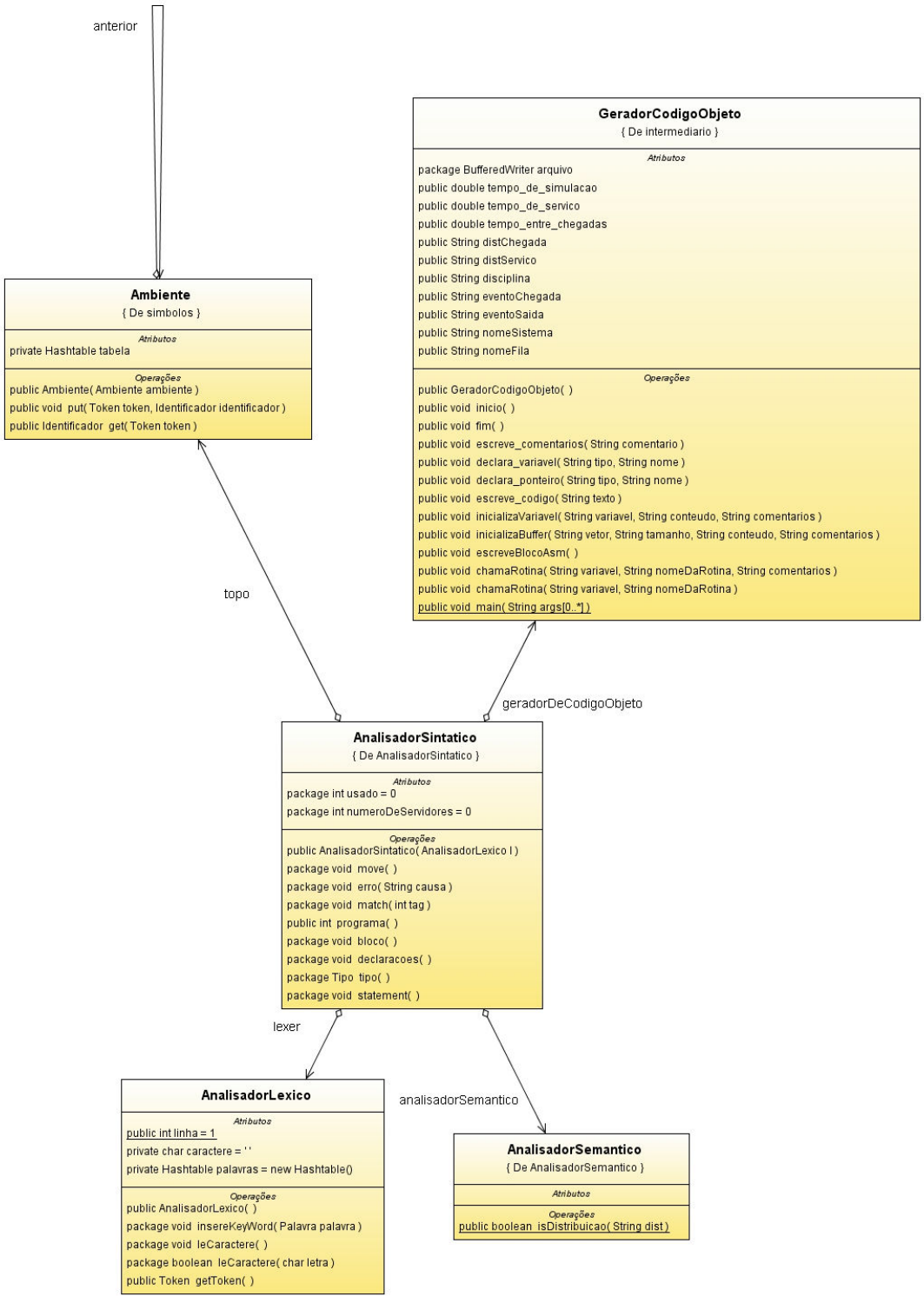


Figura 3.5 Diagrama de classes do compilador algSim

3.5.1 Análise Léxica

Na análise léxica, utiliza-se uma tabela *hash* (pacote *java.util.Hashtable*) para armazenamento das palavras reservadas da linguagem. Durante a análise léxica a cada palavra (*string*) encontrada no programa fonte, procura-se se a mesma está inserida na tabela *hash*. Se estiver inserida então é uma palavra reservada, caso contrário é uma *string* (cadeia de caracteres) que representa uma variável, ou seja, um identificador.

A seqüência de análise é realizada como descrito a seguir: as palavras reservadas da linguagem são inseridas em uma tabela *hash* antes do início da análise do programa fonte pelo analisador léxico. Concluído esse processo, inicia-se a análise léxica propriamente dita. A cada caractere (esperado) lido, o analisador léxico passa ao analisador sintático, que analisa a estrutura da sentença, procurando por possíveis erros. Caso exista algum erro, o processo de compilação é terminado e uma mensagem de erro indicando a linha em que ocorreu é ecoada na área de impressão de resultados da interface gráfica.

A seguir apresentam-se através da notação *Backus-Naur Form* (BNF) as regras para a formação de *tokens* (símbolos) na linguagem:

<palavras_reservadas> ::= algoritmo | biblioteca | defina | disciplina | eventos
| filas | fimalgoritmo | FIFO | modelo | recurso | simule
| tempo_de_servico | tempo_de_simulacao
| tempo_entre_chegadas

<comentários> ::= /*{qualquer caractere} */ | //

<caracteres_em_branco> ::= <espaço> | <tabulações>

<espaço> ::= /b

<tabulações> ::= /t

<nova_linha> ::= /n

<operador> ::= + | - | / | * |

<identificador> ::= <letra> | <identificador> [<letra> | <dígito>]

<letra> ::= a | b | c | ... | z | A | B | C | ... | Z

<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<constante_inteira> ::= {<dígito>}$^{+}$

<constante_real> ::= {<dígito>}$^{+}$.{<dígito>}$^{+}$

<constante_literal> ::= " " " {qualquer caractere ASCII} " " "

Com base na especificação da linguagem anteriormente descrita, a cada caractere lido, o analisador léxico (*lexer*) verifica se é um espaço em branco (espaços ou tabulações), se é um comentário (iniciado por /* ou //), se um caractere de nova linha (/n), se é um dígito (se for, verifica-se se é um inteiro ou real), se é uma letra (onde verifica-se se é uma palavra reservada, ou se é um identificador), se é uma constante literal - *string* (iniciada e terminada por aspas) e por fim verifica-se se é um operador da linguagem, como por exemplo (+ ou -).

Na figura 3.6 ilustra-se um trecho de código, no qual o analisador léxico reconhece e retorna para o analisador sintático (método originador da chamada), uma constante real:

```
                    ⋮
float numeroReal = valor; float potencia = 10;
for(;;){
    caractere = (char)System.in.read();
    if(!Character.isDigit(caractere))
        break;
    numeroReal = numeroReal +
            Character.digit(caractere, 10)/potencia;
    potencia = potencia * 10;
}
                    ⋮
```

Figura 3.6 Trecho de código do analisador léxico

Na figura 3.7 ilustra-se o diagrama de classes usado para a construção dos *tokens*:

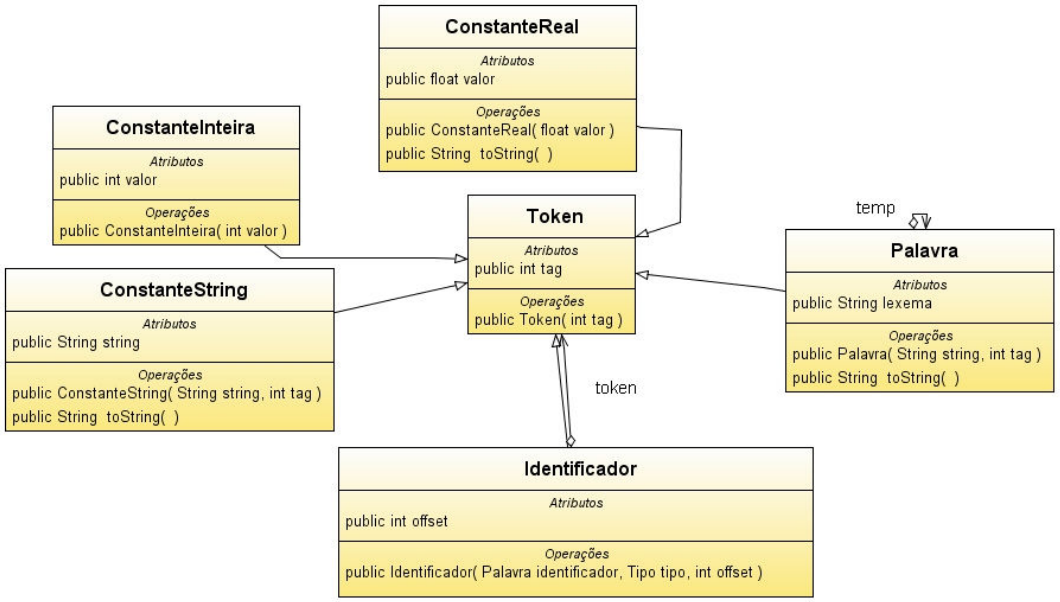


Figura 3.7 Diagrama de classes dos *tokens* da linguagem

A grande vantagem de se implementar esse analisador em JAVA é que o processo de análise é facilitado através dos métodos estáticos presentes no pacote *java.lang.Character* (*Java 2 plataform SE 5.0*), como por exemplo *Character.isDigit(Char)*, que verifica se o caractere lido é um dígito, *Character.isLetter(Char)*, que verifica se o caractere lido é uma letra (ASCII) e *Character.isLetterOrDigit*, que verifica se o caractere lido é uma letra ou dígito.

3.5.2 Análise Sintática

Dentre as técnicas de análise sintática mais comuns (*bottom-up e top-down*), a técnica adotada na implementação foi a *top-down* (análise descendente), na qual, a árvore sintática é construída da raiz para as folhas (Sebesta, 2003).

O analisador sintático implementado esboça uma árvore de análise em pré-ordem, em que o percurso nela se inicia pela raíz e cada nó da árvore é visitado na ordem esquerda-direita. Como no analisador implementado a árvore sintática é implicitamente construída, gera-se apenas o resultado do percurso dessa árvore para

cada cadeia de entrada em análise (Sebesta, 2003).

A implementação do analisador é um reconhecedor, e pelo fato de que na implementação adotada o analisador sintático conhecer todas as estruturas das sentenças da linguagem, é o único responsável pela criação de entradas na tabela de símbolos.

A gramática da linguagem (gramática preditiva LL(1)) utilizada para a construção da árvore sintática pelo analisador é ilustrada a seguir em notação **BNF** (Sebesta, 2003):

<programa> ::= **algoritmo** <identificador>
<seqüência_de_declarações>
<seqüência_de_instruções>
fimalgoritmo

<identificador> ::= <letra> | <identificador> [<letra> | <número>]

<letra> ::= a | b | c | ... | z | A | B | C | ... | Z

<número> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<seqüência_de_declarações> ::= <tipo_de_dado_modelo>
| <tipo_de_dado_recurso>
| <tipo_de_dado_defina>
| <tipo_de_dado_eventos>
| <tipo_de_dado_filas>

<tipo_de_dado_modelo> ::= **modelo** <identificador> <abre_colchetes>
{<literal_entre_aspas> <vírgula>}²<inteiro> <fecha_colchetes> **;**

<abre_colchetes> ::= " { "

<literal_entre_aspas> ::= " " "{qualquer caractere alfabético maiúsculo} " " "

<vírgula> ::= " , "

<inteiro> ::= {<numero>}$^{+}$

<fecha_colchetes> ::= " } "

<tipo_de_dado_recurso> ::= **recurso** <identificador>

<tipo_de_dado_defina> ::= **defina tempo_de_servico** <valor> **;**
| **defina tempo_de_simulacao** <valor> **;**
| **defina tempo_entre_chegadas** <valor> **;**
| **defina disciplina** <identificador> <quantidade> <abre_colchetes>

<disciplinas> <fecha_colchetes> **;**

<valor> ::= <inteiro> | <real>

<real> ::= {<inteiro>}<ponto>{<inteiro>}

<ponto> ::= " . "

<quantidade> ::= <inteiro>

<disciplinas> ::= **FIFO**

<tipo_de_dado_eventos> ::= **eventos** <identificador> <abre_colchetes>

{<caracteres_alfanuméricos_entre_aspas><vírgula>}^{2}

<caracteres_alfanuméricos_entre_aspas><fecha_colchetes>**;**

<caracteres_alfanuméricos_entres_aspas> ::= " " "{<letra> | <numero>}$^{+}$" " "

<tipo_de_dado_filas> ::= filas <identificador> <abre_colchetes>

<caracteres_alfanuméricos_entre_aspas><fecha_colchetes> **;**

<seqüência_de_instruções> ::= <instrução> <identificador> **;**

<seqüência_de_instruções> | <função> **;**

<instrução> ::= **simule**

<função> ::= **imprimeRelatorio()**

Assim, dada uma cadeia de entrada, o analisador sintático, baseando-se na descrição da sintaxe da linguagem anteriormente descrita, faz o reconhecimento de cada uma das construções aceites na linguagem desenvolvida, investigando a árvore que tem como raíz o não terminal lido e cujas folhas casam com a cadeia de entrada – *preditive parsing* (Aho *et al.*, 2007).

Desse modo, caso ocorra o reconhecimento da cadeia de entrada lida, uma árvore sintática abstrata é construída e o analisador léxico é chamado para a leitura da próxima cadeia. Caso contrário, um erro de sintaxe é emitido, conforme ilustrado no trecho de código da figura 3.8.

```
void move()throws IOException{
    //leitura do próximo token
    look = lexer.getToken();
}
void match(int tag) throws IOException{
//caractere 59 representa o caractere ponto e vírgula
    if(tag == 59)
        return;
//se o caractere lido é o caractere esperado
    if(look.tag == tag)
        move();
    else
 //trecho de código mostrado na subseção  3.5.8
        erro("Erro de sintaxe");
}
```

Figura 3.8 Trecho de código do analisador sintático

3.5.3 Análise Semântica

O analisador semântico desenvolvido tem como entrada a árvore sintática e as informações presentes na tabela de símbolos para poder então verificar a consistência semântica do programa.

No analisador semântico desenvolvido, implementou-se a verificação de escopo, verificação de tipos, verificação de distribuições estatísticas e coerção de dados de inteiro para real.

Na figura 3.9, apresenta-se um trecho de código onde se faz a verificação das distribuições presentes no código fonte.

```
public static boolean isDistribuicao(String dist){
    // verificação se a distribuição é Markov ou Determinística
    if(dist.equals("\"M\"") || dist.equals("\"D\""))
        return true;
    return false;
```

Figura 3.9 Verificação das distribuições presentes no programa fonte.

Para os tipos de dados modelo, recurso e os tipos numéricos (real e inteiro) definidos na linguagem, o trecho no qual é feita a verificação de tipos para os

mesmos é apresentado na figura 3.10.

```
public static boolean isNumerico(Tipo tipo){
    if(tipo == Tipo.Inteiro || tipo == Tipo.Real)
        return true;
    return false;
}
public static boolean isRecurso(Tipo tipo){
    if(tipo == Tipo.Recurso)
        return true;
    return false;
}
public static boolean isModelo(Tipo tipo){
    if( tipo == Tipo.Modelo)
        return true;
    return false;
}
```

Figura 3.10 Verificação dos tipos de dados

A verificação de tipos é um fator importante na confiabilidade de uma linguagem, e pelo fato da verificação de tipos em tempo de execução ser uma tarefa dispendiosa (Sebesta, 2003), neste projeto implementou-se a essa etapa em tempo de compilação (verificação estática).

3.5.4 Geração de Código Intermediário

A representação intermediária do programa fonte é uma árvore sintática abstrata ao invés de código de três – endereços. Apesar do código de três – endereços oferecer várias vantagens (em especial, facilita o particionamento de longas expressões em pequenas expressões de atribuição, facilita otimização do código gerado, etc), as árvores sintáticas abstratas fazem uma melhor representação hierárquica das estruturas sintáticas presentes no programa fonte (Aho *et al.,* 2007).

Como por opção de desenvolvimento, no código gerado não é feita nenhuma otimização independente da arquitetura, além disso, a gramática livre de contexto de algSim apresentada na seção 3.5.2 não faz qualquer uso de expressões de atribuição (o que pode ser modificado em futuras implementações, caso surja necessidade ou

desejo para tal), a técnica de código de três - endereços não teve razão de ser implementada no desenvolvimento desse módulo (gerador de código intermediário).

Em cada árvore sintática abstrata, a raiz representa um *statement*, como por exemplo, modelo, recurso, etc e os nós de cada uma das árvores representam os terminais de cada um dos *statements* (Aho *et al.,* 2007), conforme ilustrado na figura 3.11.

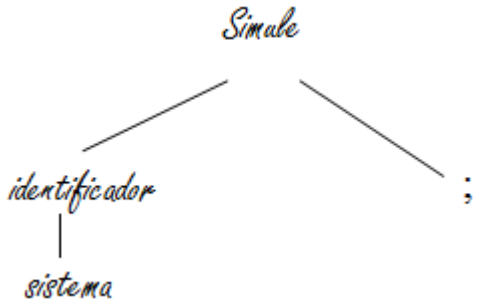


Figura 3.11 Representação intermediária do código para: “simule sistema;”

No gerador de código intermediário, a cada nó são acrescentadas informações significativas da construção (como tamanho em bytes e o tipo de dados), que são utilizadas durante o processo de geração de código. Como a abordagem escolhida na implementação do compilador é a orientação a objeto, essas informações são acrescentadas aos atributos de cada nó (objetos).

Na figura 3.12 apresenta-se um trecho de código do gerador de código intermediário.

```
void declaracoes()throws IOException{

                    ⋮
 switch(look.tag){

                    ⋮
   case Tag.RECURSO://análise da árvore com raiz recurso
        tipo = tipo();  token = look;
        match(Tag.IDENTIFICADOR);
        /*adição de novas informações ao nó
         da árvore(tipo dado)*/
        identificador =
                new Identificador((Palavra)token, tipo,usado);
        topo.put(token, identificador);
        identificador = null;
        /*adição de novas informações ao tipo de
         dado (tamanho em bytes)*/
        usado = usado + tipo.tamanho;
        break;

                    ⋮
     }
                    ⋮
 }
Tipo tipo()throws IOException{
    Tipo tipo = (Tipo)look;
    move();
        return tipo;
}
```

Figura 3.12 Trecho de código do gerador de código intermediário.

O gerador implementado também é conhecido na literatura como gerador orientado a sintaxe, atendendo que a representação intermediária é baseada na sintaxe da linguagem (Aho *et al.*, 2007).

3.5.5 Otimização de Código

Conforme citado na subseção anterior, este compilador não faz qualquer tipo de otimização independente da arquitetura na representação intermediária do código fonte. Porém, a otimização realizada é dependente da arquitetura (x86), razão pela qual ela é feita após a geração do código objeto. Esse processo é realizado na fase de

montagem e ligação através da ferramenta (compilador) *GNU C compiler* (gcc).

A vantagem oferecida pela utilização do compilador gcc para otimização do código objeto reside no aproveitamento da familiaridade da ferramenta com os detalhes mais internos das diferentes famílias de processadores da linha x86 (nomeadamente as linhas da Intel e AMD).

Dependentemente do tamanho dos registradores do processador da máquina utilizada, como por exemplo, IA-32 (Intel Architecture, 32 bit), IA-64 (Intel Architecture, 64 bits), x86-32 (AMD 32 bits), x86-64 (AMD 64 bits) e dos parâmetros utilizados durante o processo de montagem e ligação, o gcc consegue obter um melhor aproveitamento dos recursos internos oferecidos pela arquitetura desses processadores, o que pode resultar num melhor código de máquina em termos de tamanho e/ou tempo de execução (Sebesta, 2003).

A montagem e ligação são descritas mais detalhadamente na subseção 3.8.1.

3.5.6 Geração de Código

A seguir descreve-se como foram implementadas as três tarefas primárias de um gerador de código, que são: seleção das instruções, atribuição e seleção de registradores e por último a ordenação das instruções (Aho *et al.*, 2007).

A seleção das instruções é uma grande tarefa combinatória, especialmente em máquinas CISC (máquina para a qual gera-se código neste projeto), que possuem uma grande variedade de modos de endereçamento. Assim sendo, para a seleção das instruções a técnica adotada foi reescrita-da-árvore (*tree-rewriting*), que consiste na seleção das instruções com base na especificação da linguagem de alto nível, ou seja, se tomarmos como exemplo uma árvore que tem como raíz o operador "+", e temos dois registradores (operandos) como nós dessa árvore, podemos emitir a instrução add R1, R2 (Aho *et al.*, 2007).

A técnica adotada para alocação e atribuição de registradores foi atribuir valores específicos no programa fonte para determinados registradores, ou seja, a atribuição de endereços base para um determinado grupo de registradores, operações com ponto flutuante para outro grupo, os parâmetros para chamada de funções são colocados na pilha – chamada de função estilo C (Blum, 2005), etc. A vantagem

dessa técnica é que facilita o desenho do gerador de código, porém, os registradores podem ser utilizados de modo ineficiente.

Na figura 3.13 ilustra-se um trecho de código para chamada de uma função que usa os registradores de ponto flutuante:

```
__asm__(
                    ⋮
        /*
         * Chamada da routina para "setar" o tempo de simulação.
         * mas salva todos os registradores da FPU stack
         * antes de carregar o valor de tempoDeSimulação
         */
        "finit\n\t"
        "fsave buffer\n\t"
        "fldl tempoDeSimulacao\n\t"
        "pushl obj\n\t"
        "pushl e\n\t"
        "call Java_tcc_Simulacao_configuraTempoDeSimulacao\n\t"
        "addl $8,%esp\n\t"
        /*
         * Restaura os registradores da FPU
         */
        "frstor buffer\n\t"
        /*
                    ⋮

       );
```

Figura 3.13 Trecho de código para chamada de função

A técnica adotada para melhor ordenação das instruções foi deixá-las na mesma ordem em que foram produzidas pelo gerador de código intermediário (Aho *et al.*, 2007).

Conforme apresentado no código ilustrado anteriormente, a saída do gerador de código é um programa em linguagem de montagem (*ASSEMBLY*) (Blum, 2005). Dentre as várias maneiras de geração de código citadas na subseção 2.5.1, a produção de código em linguagem de montagem foi a alternativa escolhida, porque torna a geração de código um tanto mais fácil, além de ser uma alternativa razoável para máquinas com pouca memória (Aho *et al.,* 1995).

Como o programa objeto precisa de determinadas rotinas para a simulação do

modelo nele descrito (como por exemplo, rotinas para ajustar o tempo de simulação, a distribuição de chegada, a distribuição de serviço, etc) e tais rotinas estão escritas em linguagem JAVA, o código objeto (programa alvo), além de conter instruções em assembly (*inline assembly*), contém também código escrito em linguagem C (Grifith, 2002) para que através de JNI (*Java Native Interface*) (Liang,1999), troquem-se informações (ponteiros para endereços de memória, tabelas de símbolos, registradores, etc) entre os ambientes ASSEMBLY e JAVA.

Na figura 3.14 mostra-se a rotina através da qual o gerador de código objeto produz o código para a chamada de funções que usam registradores de ponto flutuante.

```
public void chamaRotina(String variavel, String nomeDaRotina)
        throws IOException{
    arquivo.write("\t"+"\"finit"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t"+"\"fsave buffer"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t"+"\"fldl "+variavel+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t"+"\"pushl obj"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t");
    arquivo.write("\"pushl e"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t");
    arquivo.write("\"call "+nomeDaRotina+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t");
    arquivo.write("\"addl $8,%esp"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
    arquivo.write("\t"+"\"frstor buffer"+"\\"+"n\\"+"t\"");
    arquivo.newLine();
}
```

Figura 3.14 Código do método do gerador de código objeto.

O assembly em linha é uma extensão do gcc que permite “misturar” código Assembly tradicional com códigos em linguagem C. Optou-se pelo uso do assembly em linha no projeto pela maior flexibilidade oferecida para trabalhar com código em linguagem C (facilita o acesso a variáveis ou funções dentro do escopo do programa) em relação ao Assembly tradicional, o que facilita a integração do código assembly

com o código C.

Além disso, a utilizade da linguagem C na codificação do arquivo objeto é indispensável para que se possa usar JNI, acrescido do fato de que, quando se trabalha com uma linguagem de baixo nível (como o assembly tradicional), é comum que se façam chamadas de sistemas para determinadas rotinas da biblioteca C, como por exemplo as rotinas para entrada e saída de dados (*scanf, read, printf, puts*, etc), leitura e escrita de arquivos (*fread, fscanf, fputs, fprintf,* etc), entre outras rotinas (visto que a linguagem assembly não oferece essas facilidades, a não ser que sejam escritas rotinas para tal, o que consome bastante tempo de desenvolvimento).

3.5.7 Tabela de Símbolos

No compilador para algSim foi projetada uma tabela de símbolos separada para cada escopo. Apesar da implementação atual da linguagem não incluir suporte para procedimentos, tal implementação é reservada para implementações futuras. A implementação deste tipo de tabelas tira proveito da regra do escopo do aninhamento mais interno (*most-closely nested rule*). O aninhamento garante que todas as tabelas de símbolos aplicáveis formem uma pilha (*stack*). No topo da pilha encontra-se a tabela de símbolos para o bloco correntemente em análise. Abaixo na pilha encontra-se a tabela de simbolo do bloco que envolve o bloco corrente e assim sucessivamente. Assim, as tabelas de símbolos podem ser alocadas e desalocadas usando as regras de manipulação de pilhas (Aho *et al.*, 2007).

Na figura 3.15 apresenta-se o código da implementação da tabela de símbolos (Aho *et al.*, 2007).

```
public class Ambiente {
    private Hashtable tabela;
    protected Ambiente anterior;
    public Ambiente(Ambiente ambiente){
        tabela = new Hashtable();
        //aninhamento de tabelas de símbolos
        anterior = ambiente;
    }
    public void put(Token token, Identificador identificador){
        //coloca o identificador na tabela de símbolos
        tabela.put(token,identificador);
    }
    public Identificador get(Token token){
        Ambiente ambiente = this;
        /*procura pelo identificador começando pela tabela
         mais interna*/
        for(; ambiente != null; ambiente = ambiente.anterior){
            Identificador achou =
                    (Identificador)(ambiente.tabela.get(token));
            if(achou != null)
                return achou;
        }
        return null;
    }
}
```

Figura 3.15 Código da tabela de símbolos (Aho *et al.*, 2007).

Adotou-se a tabela *hash* como a estrutura de dados para implementação da tabela de símbolos. Além de oferecer vantagens, como por exemplo escalabilidade, a utilização dessa estrutura (dependentemente do balanceamento de carga e da função de *hash* projetada) tem custos de pesquisa, inserção e recuperação de tempo linear (O(1)) (Cormen *et al.*, 2002).

Na linguagem Java utilizou-se o pacote *java.util.Hashtable* (*Java 2 plataform 5.0)* para criação e manipulação de tabelas *hash* (com fator de carga 0,75, que segundos os autores oferece um bom custo benefício em termos de custo de tempo e espaço).

Geralmente as informações na tabela de símbolos são inseridas pelos analisadores (léxico, sintático e semântico), mas na implementação utilizada somente o analisador sintático cria entrada na tabela de símbolos. Isto porque, atendendo o fato de que o analisador sintático conhece a estrutura sintática do programa, está em melhores condições (comparado aos outros analisadores) de distinguir entre

diferentes declarações dos identificadores e decidir se cria/usa ou não uma nova entrada na tabela de símbolos para o identificador (Aho *et al.*, 2007).

Deve-se observar que o analisador léxico também usa uma tabela durante a análise, para diferenciar entre identificadores e palavras reservadas da linguagem. Mas tal tabela contém somente as palavras reservadas da linguagem, uma vez que as mesmas não contêm nenhuma entrada na tabela de símbolos da linguagem (utilizada pelo analisador sintático).

3.5.8 Tratamento de Erros

No compilador implementado, o tratamento de erros é realizado no *front end* do compilador, ou seja, a árvore sintática passada para o *back end*, está livre de quaisquer erros e pronta para a geração de código.

Apesar de ser possível a detecção de erros durante a análise léxica (como caracteres que não formam qualquer *token* da linguagem), é na análise sintática e semântica que são detectadas a grande maioria dos erros (Aho *et al.*, 1995). Tendo em conta que no analisador semântico implementado faz-se somente a conversão implícita de tipos optou-se por realizar todo o tratamento de erros na análise sintática, já que a manipulação de todas as entradas da tabela de símbolos e a análise de fluxo de *tokens* que violem as estruturas gramaticais da linguagem (sintaxe) estão sob a responsabilidade do analisador sintático (*parser*).

Na figura 3.16 ilustra-se um método simples que foi implementado para auxiliar o analisador sintático na manipulação de erros presentes no programa fonte :

```
/**
 * Manipulação de erros:
 * Informa a linha em que o erro ocorreu e a causa do erro.
 *
 * @param causa da geração do erro
 */
void erro(String causa){
    throw new Error("Erro na linha " +
            AnalisadorLexico.linha + ": " + causa);
}
```

Figura 3.16 Código para o tratamento de erros.

3.6 Programa Objeto

O código objeto produzido é formado por uma sequência de instruções em linguagem de montagem (Assembly para x86). O programa objeto também faz referência a determinadas funções que fazem parte da biblioteca algSim, razão pela qual é necessária a inclusão do cabeçalho <*jni.h*> (definição dos tipos de dados para Java) e *"tcc_simulacao.h"* (definição dos nomes das variáveis e funções da biblioteca algSim), uma vez que as rotinas (arquivos *.class*) da biblioteca da linguagem estão escritas em linguagem JAVA.

A justificativa dos arquivos pertencentes à biblioteca algSim serem pré-compilados é que os mesmos não são necessários ao programa objeto em tempo de compilação, somente, em tempo de execução. Na figura 3.17 apresenta-se um trecho do conteúdo de um programa objeto para x86 produzido pelo compilador algSim.

```
#include <jni.h>
#include "tcc_Simulacao.h"
           ⋮
JNIEXPORT int JNICALL
Java_tcc_Simulacao_executa(JNIEnv *env, jobject objeto, jint servidores)
{
   __asm__(
           /*
            * Salva todos os registradores de propósito geral
            */
           "pushal\n\t"
           /*
            * Chamada da rotina para "setar" a disciplina da fila
            */
           "movl disciplinaFila,%edi\n\t"
           "pushl obj\n\t"
           "pushl e\n\t"
           "call Java_tcc_Simulacao_configuraDisciplina\n\t"
           "addl $8,%esp\n\t"

                       ⋮

         );

}
```

Figura 3.17 Trecho de código de um arquivo compilado na linguagem

Deve-se ressaltar que pelo fato de usar-se assembly em linha, e da passagem de ambiente Java para *Assembly* (procedimento realizado durante em tempo de execução) fez-se necessário o salvamento de todos os registradores de propósitos gerais (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), conforme mostrado no trecho de código anterior através da instrução "*pushal*", uma vez que o ambiente Java também precisa do registradores da máquina para a execução do seu código.

3.7 Biblioteca algSim

Conforme citado em seções anteriores, para a execução do programa alvo é necessário que o mesmo seja vinculado as rotinas presentes na biblioteca algSim.

Essa biblioteca foi implementada na forma de um pacote JAVA. Este pacote (módulo) é composto de diversos sub-pacotes com funcionalidades distintas e propósitos específicos. Além de simplificar o agrupamento de classes relacionadas, facilitando a localização de um determinado arquivo dentre vários outros (geralmente, vários arquivos fazem parte de um sistema de grande porte), a abordagem modular tem a vantagem de facilitar o reuso e a estendibilidade do software de acordo com necessidades futuras.

Na figura 3.18 apresenta-se o diagrama de implantação dos pacotes presentes dentro da biblioteca algSim.

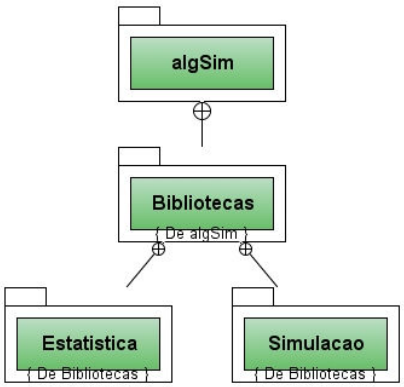


Figura 3.18 Diagrama de implantação da biblioteca algSim.

No diagrama de implantação da figura 3.18, cada um dos pacotes (diretórios), representa o seguinte:

- algSim: Pacote principal, representa a raíz dos diretórios.
- Bibliotecas: Na versão implementada, contém apenas as bibliotecas Estatística e Simulação, cujas descrições são apresentadas a seguir.
- Estatística: contém as rotinas para representação das distribuições e cálculos estatísticos da simulação. As distribuições que esta versão da biblioteca da linguagem contém são as distribuições exponencial e uniforme, cujos códigos, são ilustrados na figura 3.19.

```
public class Distribuicao {
    private final long a = 16807L;//multiplicador
    private final long m = 2147483647L;//módulo 2^31 - 1
    private long  x =  553303732L;//semente

    public double exponencial(double x){
        return -x* Math.log(geradorRandomico());
     }
     public double uniforme(double limiteInferior,
             double limiteSuperior){
         if (limiteInferior>=limiteSuperior){
             System.out.println("Erro: argumentos inválidos");
             System.exit(0);
         }
         return (limiteInferior+(limiteSuperior-limiteInferior)
                 * geradorRandomico());
     }
     public double geradorRandomico(){
          x = (a * x) % m;
         return (double)x / (double)m;
     }
}
```

Figura 3.19 Código das distribuições implementadas.

Na figura 3.20 apresentam-se dois dos vários métodos presentes na classe responsável pelos cálculos dos dados estatísticos da simulação de um programa:

```
public synchronized double getTempoMedioDeEspera( ){
    //tempo médio de espera total
    return Math.round(100.0 * (tempoTotalDeEspera /
            (double)numeroTotalDeClientes))/100.0;
}
public synchronized double getProbabilidadeDeEspera(){
    return ((double)numeroDeClientesQueEsperam /
            (double)numeroTotalDeClientes);
}
```

Figura 3.20 Trecho de código da classe que trata do cálculo dos dados estatísticos.

Os métodos são *synchronized* porque a *thread* responsável pela execução desses métodos utiliza variáveis que são compartilhadas com outras *threads*, como por exemplo, *numeroTotalDeClientes*, que é atualizada por todas as *threads* (cada *thread* representa um servidor no sistema).

Cada um dos métodos presentes nesta classe implementa uma das métricas apresentadas a seguir (Banks *et al.*,1996):

$$\text{Tempo médio de espera(min)} = \frac{\text{Tempo total que o cliente espera na fila (min)}}{\text{Número total de clientes}} \quad (3.1)$$

$$\text{Probablidade de espera} = \frac{\text{Número de clientes que esperam}}{\text{Número total de clientes}} \quad (3.2)$$

$$\text{Probabilidade de servidor ocioso} = \frac{\text{Tempo total de execução do servidor (min)}}{\text{Tempo total de execução da simulação (min)}} \quad (3.3)$$

$$\text{Tempo médio de serviço (min)} = \frac{\text{Tempo total de serviço (min)}}{\text{Número total de clientes}} \quad (3.4)$$

$$\text{Tempo médio entre chegadas (min)} = \frac{\text{Soma de todos os tempos entre chegadas (min)}}{\text{Número de chegadas} - 1} \quad (3.5)$$

$$\text{Tempo médio de espera para os que esperam(min)} = \frac{\text{Tempo total de espera dos clientes na fila(min)}}{\text{Número total de clientes}} \quad (3.6)$$

$$\text{Tempo médio do cliente no sistema(min)} = \frac{\text{Tempo total dos clientes no sistema(min)}}{\text{Número total de clientes}} \quad (3.7)$$

O pacote simulação contém o algoritmo de simulação usado e todas as estruturas (classes) auxiliares utilizadas para a execução do mesmo. Na figura 3.21 ilustra-se um diagrama que mostra o relacionamento entre algumas das classes implementadas.

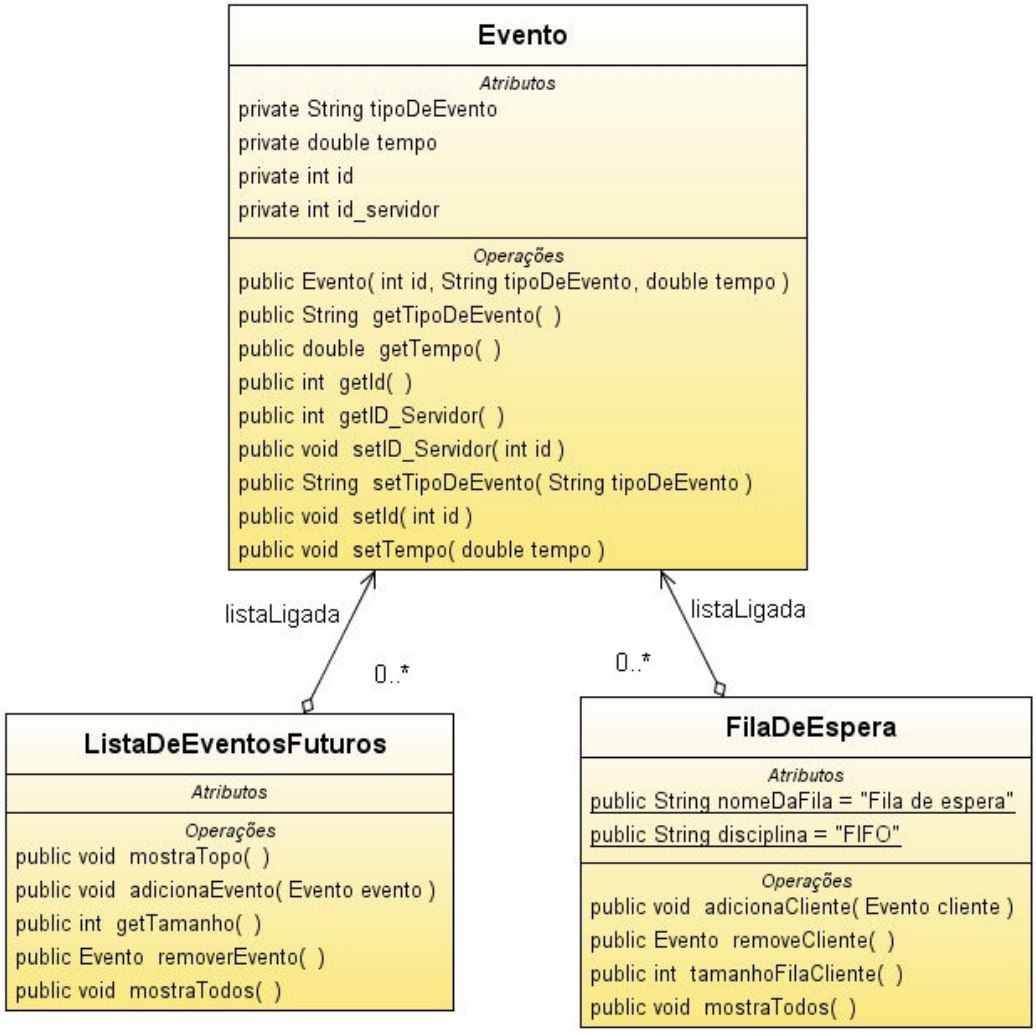


Figura 3.21 Diagrama de classes de Evento, LEF e FilaDeEspera.

Dentre as classes implementadas, encontram-se as classes ListaDeEventosFuturos.Java e FiladeEspera.Java (implementada para a política FIFO), cujos trechos de códigos são ilustrados na figura 3.22 e 3.23.

```
public class ListaDeEventosFuturos {

  static LinkedList <Evento> listaLigada =
          new LinkedList<Evento>();

                        ⋮

  public synchronized void adicionaEvento(Evento evento){
      if(listaLigada.isEmpty())listaLigada.addLast(evento);
      else{
          int i = 0;
          for(; i < listaLigada.size() && (evento.getTempo() >=
                  listaLigada.get(i).getTempo());i++);
          listaLigada.add(i, evento);
      }
  }
  public synchronized int getTamanho(){
      return listaLigada.size();
  }
  public synchronized Evento removerEvento(){
      return listaLigada.removeFirst();
  }

                        ⋮
}
```

Figura 3.22 Trecho de código da lista de eventos futuros.

```
public class FilaDeEspera {
    static LinkedList <Evento>
                  listaLigada = new LinkedList<Evento>();
                 .
                 .
                 .
    public synchronized void adicionaCliente(Evento cliente){
         listaLigada.add(cliente);
     }
     public synchronized Evento removeCliente(){
         return listaLigada.removeFirst();
     }
     public synchronized int tamanhoFilaCliente(){
         return listaLigada.size();
     }
}
```

Figura 3.23 Trecho de código da fila de espera FIFO.

Os blocos *synchronized* são blocos de código nos quais é necessário o controle de concorrência (Andrews, 2000) para execução correta (esperada) do

código, já que são códigos executados por *threads*.

O algoritmo de simulação utilizado é o *event scheduling/time-advance algorithm* (Banks *et al.*, 1996), porém, adaptou-se o algoritmo para a implementação com *threads*.

Na figura 3.24 ilustra-se o código para dar início ao carregamento das *threads*:

```
public  synchronized static void inicio_simulacao(int servidores){
    try {
        s.acquire();
        ESTADO_DO_SERVIDOR = new int[servidores];
        mutex = new Semaphore[servidores];
        for (int i = 0; i < ESTADO_DO_SERVIDOR.length; i++) {
            Simulacao.ESTADO_DO_SERVIDOR[i] = LIVRE;
            mutex[i] = new Semaphore(0);
        }
        for (int i = 0; i < servidores; i++) {
            new Simulacao(i,s).start();
        }
    } catch (InterruptedException ex) {

    }

}
```

Figura 3.24 Código para carregamento das *threads*.

Na figura 3.25 apresenta-se o código que controla a execução de cada *thread* (servidor).

```
@Override
public synchronized void run() {
    agendaProximaChegada(LEF);
    while (CLOCK <= tempoMaximo || LEF.getTamanho() > 0) {
        algoritmoDeSimulacao(LEF, fila);
        Thread.yield();
    }
    //sinaliza para a thread 0 que a thread corrente terminou.
    mutex[cont].release();
    /**
     * so a primeira Thread imprime o resultado.
     * Assim sendo, ela deve ter garantia que todas as
     * outras terminaram também.
     */
    if (cont == 0){
        try {
            for (int i = 0; i < mutex.length; i++) {
                mutex[i].acquire();
            }
            estatistica.coletaDadosEstatisticos(id);
            for(int i = 0; i < mutex.length;i++)
                mutex[i].release();
            //libera o semâforo s para o método limpaVariáveis.
            s.release();
        } catch (InterruptedException ex) {

        }
    }
}
```

Figura 3.25 Código para controle de execução das *threads*.

Como o código do algoritmo usado tem uma implementação extensa, o seu código será ilustrado em duas partes. A figura 3.26 ilustra o trecho de código do algoritmo de simulação executado no início de cada *thread* (chegada de um novo cliente).

```
public synchronized void algoritmoDeSimulacao(
            ListaDeEventosFuturos LEF, FilaDeEspera fila){
    Evento cliente,evento;
    if(LEF.getTamanho() == 0){
        agendaProximaChegada(LEF);
        return;
    }
    evento = LEF.removerEvento();CLOCK = evento.getTempo();
    // procedimento realizado quando chega um cliente
    if(evento.getTipoDeEvento().equals(GeraEventos.EVENTO_DE_CHEGADA)
            ||evento.getTipoDeEvento().equals("aguardando")){
        if(ESTADO_DO_SERVIDOR[cont] == OCUPADO){
            for(int k = 0; k < ESTADO_DO_SERVIDOR.length; k++)
                if(ESTADO_DO_SERVIDOR[k] == LIVRE){
                   LEF.adicionaEvento(evento);
                    return;
                }
            cliente = evento;cliente.setTipoDeEvento("aguardando");
            fila.adicionaCliente(cliente);
            MedidasEstatisticas.numeroDeClientesQueEsperam++;
            TAMANHO_DA_FILA = fila.tamanhoFilaCliente();
        }
        else{
            ESTADO_DO_SERVIDOR[cont] = OCUPADO;
            MedidasEstatisticas.tempoTotalDeExecucaoDoServidor +=
                    CLOCK - tempoAnterior;
            /* ajusta o id do servidor que está executando o evento*/
            evento.setID_Servidor(cont);
            /*executa o evento*/
           agendaProximaSaida(evento,LEF);
        }
        agendaProximaChegada(LEF);
    }// fim do procedimento realizado quando chega um cliente
                :
}
```

Figura 3.26 Trecho de código executado por cada *thread* na chegada de um cliente.

O código apresentando na figura 3.27 é referente ao mesmo algoritmo, porém, ilustra-se o código executado na saída de um cliente do sistema.

```
public synchronized void algoritmoDeSimulacao(
          ListaDeEventosFuturos LEF, FilaDeEspera fila){
                    .
                    .
                    .
    evento = LEF.removerEvento();CLOCK = evento.getTempo();
                    .
                    .
                    .
    /*procedimento efetuado na saída do cliente do sistema*/

    if(evento.getTipoDeEvento().equals(
                              GeraEventos.EVENTO_DE_SAIDA)){
        if(evento.getID_Servidor() != cont){
           LEF.adicionaEvento(evento);
            return;
        }
        tempoAnterior = CLOCK;
        MedidasEstatisticas.
                  tempoTotalDeExecucaoDeSimulacao = CLOCK;
        if(fila.tamanhoFilaCliente() > 0){
            cliente = fila.removeCliente();
            evento = cliente;
            evento.setID_Servidor(cont);
            agendaProximaSaida(evento,LEF);
        }
        else{
            ESTADO_DO_SERVIDOR[cont] = LIVRE;
        }

   }
}
```

Figura 3.27 Trecho de código executado por cada *thread* na saída de um cliente.

3.8 Compilador GCC

Após a geração de código, faz-se uso do compilador gcc para as etapas de montagem e ligação.

- Versão do compilador usado (saída do comando ***gcc --version***):

  gcc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu7)

  Copyright (C) 2007 Free Software Foundation, Inc.

  .

- Versão do montador utilizado (saída do comando ***as --version***):

  GNU assembler (GNU Binutils for Ubuntu) 2.18.0.20080103

Copyright 2007 Free Software Foundation, Inc.

.

- Versão do ligador (*linker*) utilizado (saída do comando ***ld --version***):
  GNU ld (GNU Binutils for Ubuntu) 2.18.0.20080103
  Copyright 2007 Free Software Foundation, Inc.

.

Para a automatização da seqüência de comandos necessários para montagem e ligação utiliza-se um arquivo Makefile (Grifith, 2002), que no código é acionado da forma apresentada na figura 3.28:

```
/**
  * Executa o script do sistema necessário
  * para a montagem e linkedição do programa
  */
public void compila(){
  barraDeProgresso.setIndeterminate(true);
  barraDeProgresso.setVisible(true);
  String saidaDoComando = null;
  try{
      labelExecutando.setVisible(false);
      barraDeProgresso.setIndeterminate(false);
      barraDeProgresso.setString("Compilando...");
      barraDeProgresso.setStringPainted(true);
      /*execução do script "make.sh" que tem como conteúdo
       o comando "make" */
      Process processo = Runtime.getRuntime().exec("sh make.sh");
      BufferedReader entradaPadrao = new BufferedReader
              (new InputStreamReader(processo.getInputStream()));
      /*laço necessário para a leitura da saída padrão
      e impressão na área de texto da interface*/
      while ((saidaDoComando = entradaPadrao.readLine()) != null)
      {
          System.out.println(saidaDoComando);
      }
      barraDeProgresso.setVisible(false);
  }
  catch(Exception e) {
      e.printStackTrace();
  }
}
```

Figura 3.28 Código para a execução do Makefile.

3.8.1 Montagem e Ligação

Nas fases de montagem e ligação utiliza-se o montador *as* (GNU *assembler*) e o ligador *ld* (GNU *linker*).

Esta etapa é necessária porque os arquivos produzidos pelo compilador algSim fazem referências a rotinas da biblioteca C e algSim.

Apesar de ambos os processos (montagem e ligação) poderem ser executados de maneira separada, utilizando-se o compilador gcc (parte da GNU *compiler collection*), podemos executar essas duas tarefas como um único processo e de modo automatizado, uma vez que o compilador gcc, além de vincular o programa objeto com as rotinas da biblioteca algSim, vincula automaticamente o código objeto com todas as bibliotecas do sistema que forem necessárias a execução do código (como por exemplo, a biblioteca C).

Apesar do processo de montagem e ligação serem executados automaticamente pela execução de um Makefile (que contém os comandos e parâmetros necessários para a execução do processo) acionado pelo compilador algSim, o processo manual (em um terminal) pode ser feito utilizando-se os seguintes comandos e parâmetros:

gcc -xc -fPIC -I jdk/include -I jdk/include/linux -shared -o programa.elf programa.obj $algSim_lib

Onde:

programa.elf: arquivo de saída (programa executável)

programa.obj: programa objeto (código objeto do programa desenvolvido pelo usuário).

$algSim_lib: caminho no sistema para acesso as rotinas da biblioteca (pacote) algSim, tais como, libsetDistribuicaoChegada.so, libsetDistribuicaoServico.so, libsetTempoServico.so, entre outras rotinas.

3.9 Programa Executável

O programa executável é gerado em formato *ELF* (*Executable and Linking Format*), que é o formato padrão dos arquivos executáveis em ambientes UNIX e derivados.

Este é o arquivo que é carregado na memória pelo sistema operacional e contém todos os vínculos com as bibliotecas necessárias para a sua execução.

Na figura 3.29 mostra-se o trecho de código que faz a chamada para a execução do programa em formato *ELF* (arquivo binário – código de máquina). O uso do semáforo é necessário por causa da concorrência entre os métodos executa(int) e limpaVariáveis(). Tal concorrência ocorre pelo fato do método limpaVariaveis ser estático e do método executa(int) iniciar uma ou várias *threads*, segundo o valor do parâmetro *int*, ou seja, a ordem de execução dos mesmos depende do escalonamento do sistema operacional e não da ordem de chamada no programa.

```
/**
 * cria uma instância da classe Simulação
 * e inicia a execução da mesma
 */
public void chamaSimulacao(){
  System.out.println(" Executando:");
  Semaphore p = new Semaphore(1);
  //servidores representa o número de threads
  new Simulacao(1,p).executa(servidores);
  Simulacao.limpaVariaveis();
}
```

Figura 3.29 Código para a execução do programa de simulação

3.10 Saída da simulação

A saída da simulação é produzida no final da execução do programa executável. Apesar de também poder ser impressa num terminal, decidiu-se reservar uma área na interface gráfica para apresentação dos resultados da execução do

programa, tirando-se proveito do fato de ter-se desenvolvido uma interface gráfica para a edição, compilação e execução dos programas da linguagem.

Na figura 3.30 ilustra-se o código de uma das classes responsáveis pela impressão de dados na área de impressão da interface (editor de código).

```
class Console{
    PipedInputStream canalDeEntrada;
    PipedOutputStream canalDeSaida;
    JTextArea terminal;
    public Console(JTextArea areaDeTexto) throws IOException {
        // "Seta" System.out
        canalDeEntrada = new PipedInputStream();
        canalDeSaida = new PipedOutputStream(canalDeEntrada);
        System.setOut(new PrintStream(canalDeSaida, true));
        terminal = areaDeTexto;
        terminal.setEditable(false);
        terminal.setRows(20);
        terminal.setColumns(70);
        // Cria as threads leitoras
        new ReaderThread(canalDeEntrada).start();
    }
```

Figura 3.30 Código de uma das classes necessárias para a impressão na interface

3.11 Considerações Finais

Neste capítulo foram abordados os detalhes de implementação do ambiente de programação da linguagem (o editor, o compilador e própria linguagem).

No desenvolvimento da linguagem, foram levados em consideração critérios como a legibilidade (simplicidade na expressão dos algoritmos), a capacidade de escrita (pouco número de construções, suporte para abstrações e expressividade), a confiabilidade (verificação de tipos) e o custo global (facilidade no aprendizado da linguagem e na escrita de programas), que estão entre os parâmetros mais importantes para avaliar uma linguagem de programação (Sebesta, 2003).

A portabilidade da ferramenta aqui desenvolvida é conseguida por meio da linguagem Java (utilizada no desenvolvimento da interface gráfica, do compilador e da biblioteca algSim). Quanto ao código objeto gerado, a sua portabilidade depende somente dos compiladores C (gcc - Linux, CygWin - Windows) disponíveis para as

arquiteturas x86 (independentemente de serem 32 ou 64 bits).

No próximo capítulo relatam-se os resultados dos testes realizados no ambiente de programação desenvolvido.

Capítulo 4 – Testes e Validação

4.1 Considerações Iniciais

Neste capítulo apresentam-se os testes realizados no compilador algSim. Para os testes, foi utilizado um único arquivo, porém o mesmo possui duas versões , uma sem qualquer erros (programa correto) e outra com vários erros (um erro léxico, um sintático e um semântico).

O arquivo contendo erros foi utilizado para verificar o funcionamento do *front end* do compilador. Por outro lado, o arquivo correto é utilizado para verificar o funcionamento do *back end* do compilador e para a obtenção dos resultados esperados através da simulação do programa.

4.2 Arquivo de Testes

O arquivo de teste utilizado representa o modelo de funcionamento do processador de um computador. O recurso compartilhado é o processador do sistema. Assume-se que a máquina utilizada possui dois núcleos (tecnologia atualmente conhecida como Dual core/Core 2 Duo). No sistema os clientes representam os processos do(s) usuário(s), utiliza-se uma fila única que representa a fila de processos prontos no sistema. A política (disciplina) de escalonamento utilizada pelos processadores do sistema é FIFO. A taxa de chegada (criação) de processos no sistema é de um (1) cliente a cada cinco (5) minutos e é representada pela distribuição exponencial. Para todos os processos do sistema adotou-se um tempo de serviço (tempo de CPU) de cinco (5) minutos, representado no modelo pela distribuição exponencial. O sistema descrito é simulado por sessenta (60) minutos.

A figura 4.1 ilustra a rede de fila simulada:

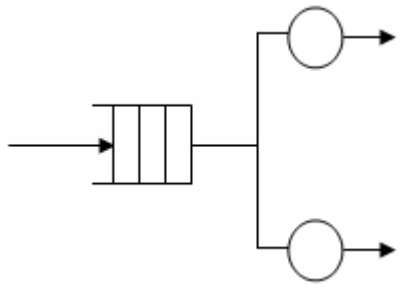
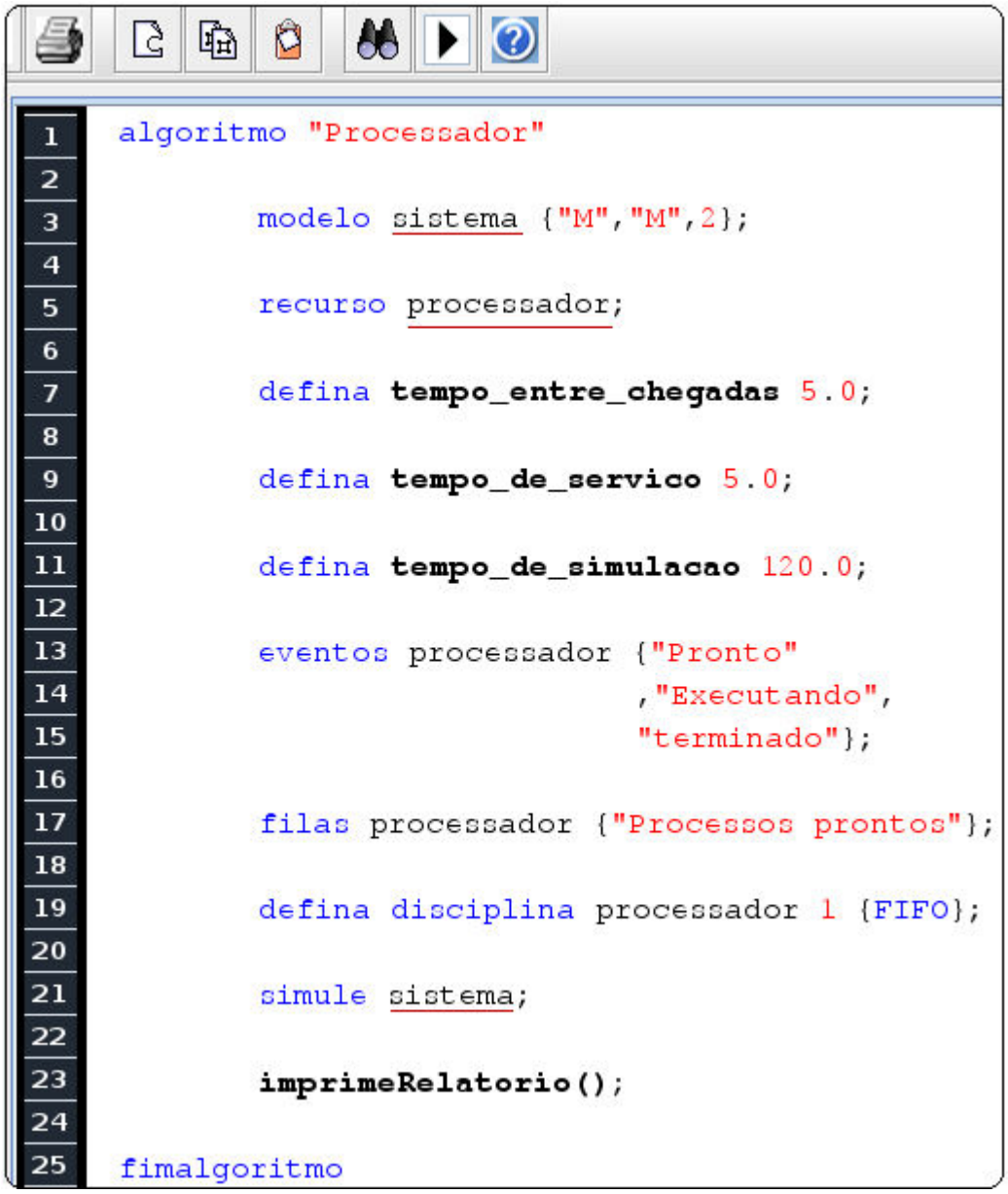


Figura 4.1 Rede de fila (M/M/2) representada pelo programa processador.alg

A figura 4.2 ilustra o arquivo de testes usado para os testes e validação do projeto desenvolvido. Este arquivo encontra-se livre de quaisquer erros de compilação. Porém as palavras sublinhadas em vermelho (linhas 3, 4 e 21), são os pontos do programa em que serão inseridos os erros léxicos, sintáticos e semânticos.



```
1  algoritmo "Processador"
2
3          modelo sistema {"M","M",2};
4
5          recurso processador;
6
7          defina tempo_entre_chegadas 5.0;
8
9          defina tempo_de_servico 5.0;
10
11         defina tempo_de_simulacao 120.0;
12
13         eventos processador {"Pronto"
14                              ,"Executando",
15                              "terminado"};
16
17         filas processador {"Processos prontos"};
18
19         defina disciplina processador 1 {FIFO};
20
21         simule sistema;
22
23         imprimeRelatorio();
24
25 fimalgoritmo
```

Figura 4.2 Arquivo de testes (processador.alg)

4.3 Detecção de Erros pelo *Front End*

Em cada uma das três (3) subseções seguintes, apresenta-se como o compilador desenvolvido lida com cada tipo de erro. Para a detecção dos erros, adotou-se a técnica de testes de caixa branca (*White box testing*) (Pressman, 2005).

4.3.1 Erros Léxicos

No código mostrado na figura 4.2, a linha três (3) foi alterada com o objetivo de introduzir-se um erro léxico no arquivo processador.alg. A figura 4.3, ilustra apenas o novo conteúdo da linha alterada:

```
modelo 1sistema {"M","M",2};
```

Figura 4.3 Introdução de erro léxico no arquivo.

A palavra "1sistema" é vista como um erro pelo analisador léxico, porque a sua forma não corresponde a nenhuma regra de formação de *tokens* na linguagem algsim. Assim, o analisador após a leitura desta cadeia de caracteres (1sistema), emite o erro mostrado na figura 4.4.

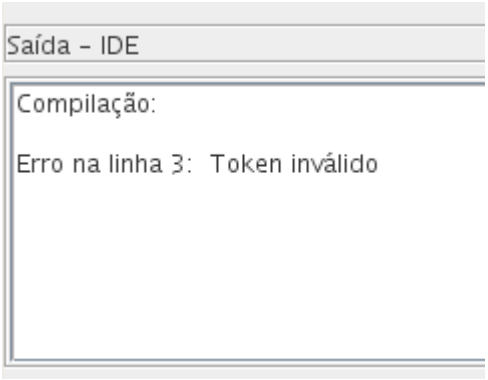


Figura 4.4 Resultado da compilação de um erro léxico.

4.3.2 Erros Sintáticos

Para o teste do analisador sintático, alterou-se o conteúdo da linha cinco (5), com o objetivo de introduzir-se um erro de sintaxe no arquivo processador.alg. O novo conteúdo dessa linha é mostrado na figura 4.5.

```
recurso "Processador";
```

Figura 4.5 Introdução de um erro sintático no arquivo.

O resultado produzido pelo analisador sintático é mostrado na figura 4.6. A emissão de tal mensagem de erro deve-se ao fato do analisador sintático estar esperando um identificador ao invés de uma constante literal (*string*).

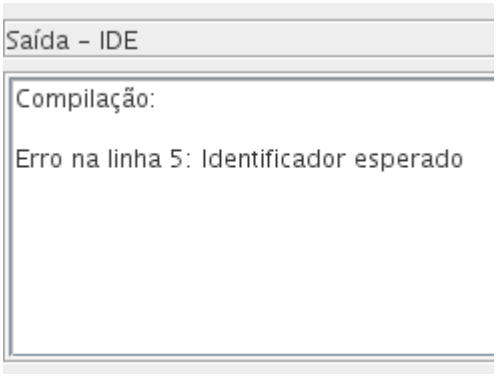


Figura 4.6 Resultado da análise de um erro sintático.

4.3.3 Erros Semânticos

Um erro semântico foi introduzido na linha vinte e um (21) do arquivo processador.alg, ou seja, o conteúdo da linha vinte e um foi alterado conforme mostrado na figura 4.7.

```
simule CPU;
```

Figura 4.7 Introdução de um erro semântico no arquivo.

Como o identificador CPU não foi previamente declarado, implica que o mesmo não se encontra presente na tabela de símbolos. Assim sendo, um erro semântico é emitido (durante a verificação de escopo), conforme mostrado na figura 4.8.

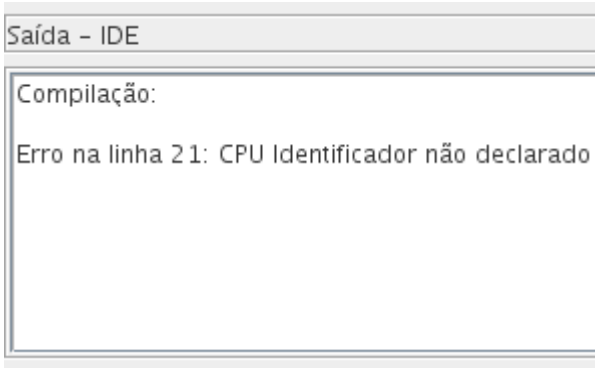


Figura 4.8 Resultado da compilação de um erro semântico.

4.4 Resultados da Simulação

Os resultados produzidos em da simulação feita com o arquivo processador.alg (arquivo ilustrado na figura 4.2) são apresentados na figura 4.9.

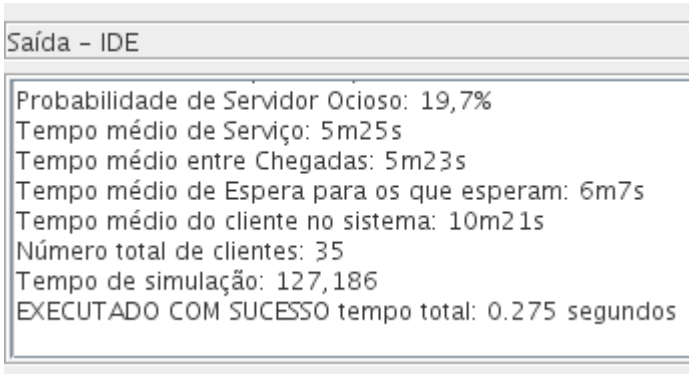


Figura 4.9 Resultados da simulação de processador.alg (modelo M/M/2).

Todos os dados apresentados na saída da execução do programa de simulação, exceto a penúltima e a última linha, são os resultados retornados por cada um dos métodos que implementam cada uma das métricas mostradas na seção 3.7. A penúltima linha apresenta o tempo total de execução da simulação (relógio da simulação) e a última linha mostra o tempo de máquina para a execução do programa processador.alg.

4.5 Validação de algSim

Para a validação dos resultados obtidos neste projeto, comparam-se os resultados obtidos com o SMPL (MacDougall, 1987) e os de algSim. Os programas simulados em smpl apresentaram os mesmos resultados quando simulados em algSim.

As métricas de smpl utilizadas na comparação com os resultados de algSim são a utilização do servidor (*UTILIZATION*), tempo médio de ocupação do servidor (*MEAN BUSY PERIOD*), e o numero de liberações do recurso (*RELEASE*). Em algSim as medidas equivalentes são a Probabilidade de servidor ocioso – cujo o complemento corresponde a *UTILIZATION* (em termos de probabilidade) do servidor (Banks *et al.,* 1996), o Tempo médio de serviço – que corresponde a *MEAN BUSY PERIOD* e o Número de clientes – que pela lei de Little e assumindo-se o fluxo balanceado (*flow balance assumption*) corresponde a *RELEASE*.

4.6 Considerações Finais

Neste capítulo foram apresentados os testes realizados do projeto desenvolvido, no qual inicialmente testou-se o arquivo de entrada processador.alg com erros detectáveis durante a fase de análise e finalmente testou-se o mesmo arquivo livre de quaisquer erros. Apresentaram-se também os resultados dos testes realizados com o arquivo de entrada e por último abordou-se o procedimento utilizado para a validação do projeto algSim (utilização da ferramenta SMPL para a construção de programas equivalentes aos construídos em algSim e para a comparação dos resultados das métricas comuns entre SMPL e algSim).

Capítulo 5 – Conclusões

5.1 Considerações Iniciais

Neste capítulo apresentam-se as conclusões feitas sobre o projeto, as dificuldades encontradas ao longo do desenvolvimento do mesmo e propostas para projetos futuros.

5.2 Dificuldades Encontradas

A grande dificuldade encontrada no desenvolvimento do projeto foi a especificação da linguagem algSim, de modo que a linguagem resultante fosse simples e intuitiva. As outras dificuldades estiverem relacionadas à abstração da implementação do relógio de simulação, escalonamento de eventos, controle de concorrência das *threads* na implementação do algoritmo de simulação e construção do *back end* do compilador para geração de código para arquitetura x86.

5.3 Conclusões

No projeto algSim, além de desenvolver-se uma interface gráfica (editor de códigos com o recurso *syntax highlighting*), um *back end* para a linguagem e gerar-se código executável para a arquitetura x86 (partes não implementadas em LisRef), reestruturou-se o *front end* e a gramática da linguagem devido a abstração de várias construções na linguagem algSim que antes eram necessárias em LisRef (Oliveira, 2006) para a simulação de um programa. Essas abstrações simplificaram a quantidade de declarações de variáveis de simulação no programa, o que reduziu significativamente o tamanho do arquivo fonte (código fonte) e o número de construções que um usuário da linguagem algSim deve aprender. As construções usadas na linguagem algSim são bastantes próximas ao pseudocódigo, além disso abstraíram-se as construções na linguagem LisRef para controle do relógio de simulação, controle do escalonamento, inserção/remoção de eventos no sistema (na LEF e na fila de espera), liberação dos recursos compartilhados e outras construções que requerem o uso de estrutura de repetição (ENQUANTO) e condicional (SE...SENAO...FIMSE), visto que são algumas das várias estruturas que causam a maior parte das dificuldades no aprendizado de uma linguagem de programação por parte dos usuários.

5.4 Propostas para Trabalhos Futuros

Como proposta para trabalhos futuros, sugere-se a introdução de construções na linguagem que possibilitem o usuário simular modelos mais complexos. Uma outra medida é o suporte a mais funções de distribuição de probabilidade, como a distribuição normal, k-erlang, hiper-exponencial entre outras, além das distribuições exponenciais (M) e uniformes (D) já implementadas na linguagem.

Para as construções na linguagem, sugere-se a criação de construções que permitam a união de modelos em série ou paralelamente, como por exemplo, fazer combinações de modelos M/M/K com modelos M/D/K, D/D/K, e outras combinações destes.

Uma outra melhoria que representará um recurso bastante significativo na linguagem é o suporte a várias políticas de escalonamento além da FIFO que é a única implementada na versão atual.

Referências Bibliográficas

(Andrews, 2000) ANDREWS, G. R., *Foundations of Multithreaded, Parallel, and Distributed Programing*, Addison Wesley, 1999.

(Ascencio & Campos, 2003) ASCENCIO, A. F. G., CAMPOS, E. A. V. , *Fundamentos da Programação de Computadores: Algoritmos, Pascal e C/C++*, Prentice Hall, 2003.

(Aho *et al.*, 1995) AHO, A. V., SETHI R., ULLMAN J. D*., Compilers: Principles, Techniques and Tools*, Addison Wesley, 1987.

(Aho *et al*., 2007) AHO, A. V., LAM, M. S., SETHI R., ULLMAN J. D*., Compilers: Principles, Techniques and Tools*, 2ª edição, Addison Wesley, 2007.

(Balieiro, 2005) BALIEIRO, M. O. S., *Protocolo Conservativo CMB para Simulação Distribuída,* Dissertação de Mestrado, DCT – UFMS, 2005.

(Banks *et al.*, 1996) BANKS, J., CARSON, J. S., NICOL, D. M., NELSON B. L., *Discrete-Event System Simulation*, 3ª edição, Prentice-Hall, International Series In Industrial and Systems Engineering, 2001.

(Blum, 2005) BLUM, Richard., *Professional Assembly Language*, Wiley Publishing Inc, 2005.

(Cardoso & Vallete, 1996) CARDOSO, J., VALLETE, R., *Redes de Petri,* UFSC, 1997.

(Cormen et al., 2002) CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., *Algoritmos*, 2ª edição, Campus, 2002.

(Deitel & Deitel, 2004) Deitel, H.M., Deitel, P.J., *JAVA How to Program*, 6ª edição, Prentice Hall, 2004.

(Forbellone, 1999) FORBELLONE, A. L. V., EBERSPACHER, H. F., *Lógica de Programação*, Makron Books, 1999.

(Gonçalves, 2006) GONÇALVES, E., *Dominando Netbeans*, Ciência Moderna, 2006.

(Ghezzi, 1991) GHEZZI, C., JAZAYERI, M., *Conceitos de Linguagens de*

Programação, Campus, 1991.

(Griffith, 2002) GRIFFITY, A., *GCC: The Complete Reference,* McGraw-Hill/Osborne, 2002.

(Larman, 2004) LARMAN, C., *Utilizando Uml e Padrões, 2ª edição, Bookman, 2004.*

(Liang, 1999) LIANG, S., *The Java Native Interface Programmer's Guide and Specification*, Addison Wesley, 1999.

(MacDougall, 1987) MACDOUGALL, M. H., *Simulating Computer Systems Techniques and Tools*, The MIT Press, 1987.

(Marcari & Manacero, 2003), MARCARI JUNIOR, E., MANACERO JUNIOR, A., *Classificação de Técnicas para Análise de Desempenho de Sistemas Paralelos e Distribuídos,* Notas didáticas do IBILCE, IBILCE – UNESP, *2003.*

(Oliveira, 2006) OLIVEIRA, V.G., *Uma linguagem algorítmica para simulação de redes de filas*. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Biociências, Letras e Ciências Exatas, Universidade Estadual Paulista, São José do Rio Preto, 2006.

(Santana *et al.*, 1994) SANTANA, R. H. C., SANTANA, M. J., ORLANDI, R. C. G. S., SPOLON, R., Júnior, N. C., *Técnicas para Avaliação de Desempenho de Sistemas Computacionais*, Notas didáticas do ICMC, ICMC – USP, 1994.

(Sebesta, 2003) SEBESTA, R. W., *Conceitos de Linguagens de Programação*, 5ª edição, Bookman, 2003.