Linguagem Algorítmica para Simulação de Redes de Filas

Renata Spolon Lobato[1], Vanessa Gomes de Oliveira[1] e Roberta Spolon Ulson[2]

[1]Instituto de Biociências, Letras e Ciências Exatas, UNESP – Universidade Estadual Paulista, São José do Rio Preto, SP, Brasil.
renata@ibilce.unesp.br, vanessa_comput@yahoo.com.br

[2]Faculdade de Ciências, UNESP – Universidade Estadual Paulista, Bauru, SP, Brasil.
roberta@fc.unesp.br

Resumo. Este trabalho apresenta uma linguagem algorítmica de programação, baseada em português estruturado, concebida para construir programas de simulação de redes de filas. Esta linguagem foi fundamentada na biblioteca RFOO e possui comandos e estruturas que são de fácil compreensão, buscando permitir que um usuário com conhecimentos básicos sobre programação e simulação de redes de filas possa implementar programas de simulação sem a necessidade de construir as estruturas necessárias. A versão atual do compilador da linguagem gera código binário para a família de processadores Nios II, mas a geração de código binário para a arquitetura x86 também é possível.

Palavras-chave: compilador, simulação, redes de filas.

1 Introdução

O interesse crescente no desenvolvimento da área de simulação ocorre principalmente devido ao aumento da complexidade dos problemas a serem resolvidos e à constante necessidade por ferramentas de avaliação de desempenho [5]. As ferramentas de avaliação de desempenho dividem-se em técnicas de aferição (prototipação, *benchmarks* e coleta de dados) e técnicas de modelagem (solução analítica e simulação). As técnicas de aferição são aplicadas quando o sistema ou um protótipo já existe e as medidas de desempenho são realizadas sobre o próprio sistema, enquanto as técnicas de modelagem são utilizadas quando o sistema não existe, está em fase de desenvolvimento ou não pode ser testado. Para a utilização das técnicas de modelagem torna-se necessário o desenvolvimento de um modelo do sistema a ser avaliado [3]. Na solução analítica, a modelagem é feita em termos de equações, e o acréscimo de novas características no modelo pode aumentar a complexidade da solução. Na simulação, o modelo é transformado em um programa que representa o sistema real.

A avaliação de desempenho pode ser efetuada em tipos variados de sistemas e situações, incluindo: quando o sistema real não existe e deseja-se conhecer o seu comportamento futuro; quando a experimentação com o sistema real tem um alto

custo e a simulação pode ser uma alternativa mais realista do comportamento do sistema; quando efetuar experimentos com o sistema real não é apropriado ou mesmo quando isto pode exigir o aniquilamento do próprio sistema [6].

Ao optar-se por utilizar as técnicas de simulação para resolver o modelo de um sistema, são necessárias as estruturas que fornecem suporte para a simulação: a criação de uma lista de eventos futuros, que contém os eventos a serem realizados no sistema, tais como a chegada de um novo elemento ao sistema; um relógio global que irá controlar a passagem do tempo da simulação; algumas variáveis que irão descrever o estado do sistema e um gerador de números aleatórios que irá gerar números aleatoriamente para a manipulação do tempo do sistema, sendo imprescindíveis para se executar a simulação.

Com isso, o trabalho do programador para construir todas as estruturas necessárias pode atrapalhar a criação do modelo de filas ou até mesmo todo o estudo de desempenho. Sendo assim, este trabalho propõe o uso de uma linguagem algorítmica para a simulação de redes de fila, que dispensa a construção das estruturas já mencionadas, denominada LiSReF (**L**inguagem para **Si**mulação de **Re**des de **F**ila) [11]. Esta linguagem foi baseada na biblioteca RFOO (Biblioteca de **R**edes de **F**ila **O**rientada a **O**bjetos) [4], cuja função é permitir que o usuário utilize a simulação de redes de filas sem a necessidade de construir todas as estruturas e métodos necessários.

A arquitetura alvo da LiSReF é o processador Nios II [2], um processador virtual do tipo RISC (*Reduced Instructions Set Computer*) que pode ser configurado pelo usuário (incluindo os periféricos).

Este trabalho está organizado da seguinte forma: na seção 2 é descrita a biblioteca RFOO; na seção 3 são descritos trabalhos relacionados à LiSReF; na seção 4 é apresentada a descrição da linguagem, abordando a estrutura básica de um programa escrito em LiSReF, as definições para as análises léxica, sintática e semântica e a geração de código, além de um programa escrito nessa linguagem e na seção 5 são exibidas as conclusões do trabalho.

2 Biblioteca RFOO

A biblioteca RFOO auxilia o programador na elaboração de um ambiente de simulação de redes de filas, uma vez que ela possui todas as estruturas necessárias para o mesmo. A RFOO foi construída com a linguagem de programação C++ e é formada por classes divididas em pacotes que fazem a construção do ambiente de simulação [4]:

- Pacote Centro de Serviço: é responsável pelo controle dos centros de serviço do sistema, formado por seus respectivos servidores e filas;
- Pacote Lista de Eventos Futuros: é uma estrutura que controla a ordem de execução dos eventos da simulação que está sendo executada. Quando ocorre um evento, ele é retirado desta lista e o relógio global do sistema é atualizado;
- Pacote Estatística: realiza os cálculos estatísticos do sistema, relativos aos servidores considerando um único usuário ou todos os usuários;

- Pacote Relógio: é formado pelas classes Relógio e Aleatório. A primeira controla o relógio do sistema, monitorando a passagem de tempo da simulação. A classe Aleatório realiza a geração de números pseudo-aleatórios, segundo algumas distribuições de probabilidade.

3 Trabalhos Relacionados

Uma das formas de se implementar um programa de simulação é através de linguagens de simulação. Estas linguagens contêm todas as estruturas necessárias para a criação de um ambiente de simulação, livrando o programador da necessidade de executar tal tarefa. Como exemplo de uma linguagem deste tipo cita-se SIMSCRIPT III [12].

As linguagens algorítmicas são principalmente utilizadas por usuários com pouco conhecimento sobre programação em cursos introdutórios. Elas possibilitam de maneira clara a elaboração da solução computacional, sem que o usuário tenha que se preocupar com detalhes de uma linguagem de programação. Genesis [9], por exemplo, é uma linguagem algorítmica dirigida a programadores iniciantes, com o intuito de auxiliar na resolução de problemas computacionais e no desenvolvimento da habilidade de construir algoritmos [10].

Embora as linguagens de simulação possuam todas as estruturas para a construção de um programa de simulação, elas exigem que o programador conheça devidamente a sua sintaxe. Assim, para evitar que o usuário tenha que aprender uma nova linguagem de programação complexa, a LiSReF aproveita as vantagens do uso de linguagens algorítmicas. Além disso, o usuário não precisa construir as estruturas necessárias para a simulação de filas.

4 A Linguagem Algorítmica para Simulação de Redes de Fila

A estrutura da linguagem algorítmica LiSReF tem o objetivo de fazer com que o programador preocupe-se com a definição do modelo, dos parâmetros e da correta implementação do programa de simulação correspondente. Dessa forma, o programador não precisa cuidar dos detalhes referentes à elaboração das estruturas fundamentais para a execução de uma simulação. Vale lembrar que, para o atendimento dos usuários das redes de filas da LiSReF foi definida a política de escalonamento FCFS (*First Come First Served*).

Na simulação discreta, abordagem utilizada para a simulação na LiSReF, as mudanças no estado do sistema em estudo ocorrem nos tempos de eventos. Devido a essa característica, pode-se ter uma descrição completa do estado do sistema através do avanço do tempo entre um evento e outro. A LiSReF permite a implementação de um programa de simulação através da definição das mudanças que podem ocorrer nos estados, a cada tempo de evento [3].

4.1 Estrutura de um Programa em LiSReF

Um programa escrito em LiSReF é dividido em duas partes. A primeira contém a declaração das variáveis e a segunda a seqüência de comandos. A estrutura básica de um programa em LiSReF é apresentada na Figura 1.

```
DECLARAÇÕES
declarações de variáveis
FIMDECLARAÇÕES
PROGRAMASIMULAÇÃO
comandos da linguagem e execução do programa
FIMDASIMULAÇÃO
```

Figura 1. Estrutura de um programa em LiSReF.

4.2 Análise Léxica

Para a construção do módulo de análise léxica foi utilizada a ferramenta *Flex* [8]. Foi criado um arquivo que contém as especificações da gramática regular da linguagem, ou seja, os *tokens* que podem ser reconhecidos pela LiSReF. Este arquivo, depois de compilado pelo *Flex*, gera código a ser utilizado pelo analisador sintático, uma vez que ele retorna ao sintático os *tokens* encontrados no arquivo de entrada.

A construção da parte correspondente à especificação da gramática da LiSReF foi realizada através da elaboração das definições regulares dos *tokens* a serem reconhecidos, como pode ser observado na Tabela 1. O símbolo "+" depois de uma definição entre colchetes significa "uma ou mais" ocorrências, o símbolo "*" representa "zero ou mais" ocorrências e o símbolo "?" significa "uma ou nenhuma" ocorrência da definição precedente.

O analisador léxico também engloba um tratador de erros, construído com a linguagem C, que se encarrega de identificar os erros presentes nesta fase da compilação. Os erros que este analisador trata são relativamente simples, devido à natureza da análise léxica, e incluem as construções inválidas identificadas ao longo do programa fonte (qualquer forma de representação que não obedeça a uma das especificações possíveis da LiSReF [11]). Assim que um erro é identificado ele é inserido na lista de erros léxicos. Caso esta lista possua elementos, ela é exibida ao usuário ao final da fase de *front-end*.

4.3 Sintaxe da Linguagem

A LiSReF define as construções básicas de uma linguagem de programação imperativa, que inclui comandos para repetição, comando condicional, comando de atribuição e operadores lógicos, aritméticos e relacionais. Na construção do analisador sintático foi utilizada a ferramenta *Bison* [8], que é própria para a construção deste tipo de analisador. A diferença, quando comparada com outras

linguagens algorítmicas de programação, está no conjunto de instruções que permitem a manipulação de estruturas e construções de simulação de redes de fila:

- Para a declaração de um identificador do tipo centro de serviço: *CriaCentrodeServiço id, númeroInteiro*;
- Para a declaração de um identificador relacionado ao relógio do sistema: *CriaRelógio id*;
- Para a manipulação das estatísticas: *CriaEstatística id*. Cria um identificador que se relacionará a um centro de serviço, para a geração de seus dados estatísticos. Exemplos desses dados incluem o tamanho médio da fila e o tempo médio que um cliente aguarda ser atendido;
- Para se determinar o evento inicial do sistema: *EventoInicial nomeCds (nomeEvento)*;
- Para atribuir o tempo atual do sistema à variável de simulação *TempoAtual*: *TempoAtual <- RecebeTempoAtual (id)*, em que *id* é o identificador relacionado ao relógio do sistema;
- Para fazer a liberação de um servidor: *SaiDoServidor nomeCds (nomeEvento).*
- Para simular um evento em um tempo determinado: *Simular nomeEvento(Usuário, tempo, próximoEvento)*;
- Para retornar o estado corrente de um servidor: *RetornaEstadoServidor (nomeCds)*: retorna DESOCUPADO se o servidor do centro de serviço *nomeCds* estiver livre e OCUPADO, caso contrário. Seu valor deve ser relacionado à variável EstadoServidor da seguinte maneira: *EstadoServidor <- RetornaEstadoServidor (nomeCds).*

Tabela 1. *Tokens* e definições regulares da LiSReF [11].

Token	**Definição Regular**
Identificador: *id*	[a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+
Número inteiro: *numint*	[-+]?[0-9]+
Número real: *numreal*	[-+]?[0-9]+[.][0-9]+
Comentário	coment #.*
Caracteres especiais	( ) [ ] ; ,
Operadores aritméticos	+ - * /
Operadores lógicos	E OU
Operadores de comparação	<, <=, >, >=, <> (diferente), =
Caracteres dispensáveis	eb [/t/r]+ (espaços em branco e tabulação)
Nova linha: *nl*	[\n]
Palavras que se iniciam com letras, mas possuem dígitos em sua constituição (inválidos)	[a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+[0-9]+ [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ0-9]*
Símbolos que se iniciam com dígitos, mas que possuem alguma letra em sua constituição (inválidos)	[0-9]+ [a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]+[0-9 a-zçáéíóúàãõA-ZÇÁÉÍÓÚÀÃÕ]*
String delimitada por aspas	\"[^\" \n]*[\" \n]

O analisador sintático também possui um tratador de erros, construído com a linguagem C [13]. Quando este analisador encontra uma construção inválida para a LiSReF ele adiciona as características deste erro a uma lista, tal como o analisador léxico. Esta lista também será exibida ao usuário ao final da *front-end*.

A forma escolhida para o tratamento de erros foi a que utiliza palavras-chave ou símbolos de terminação de frase como pontos de sincronização da análise do programa, método chamado modalidade do desespero [1]. Este método permite que a análise sintática continue e, caso algum erro seja encontrado o analisador descarta símbolos até encontrar o próximo ponto-e-vírgula. Assim, o sintático pode começar a analisar novamente o programa depois que um erro é identificado. Para construções da linguagem, o analisador faz a sincronização a partir dos símbolos delimitadores da LiSReF, e que inclui, por exemplo, as palavras COMEÇO, FIMSE e FIMENQUANTO.

4.4 Semântica da Linguagem

A última fase de análise da *front-end* consiste no analisador semântico. Este analisador foi construído na linguagem C [13] e com o compilador *gcc* do ambiente Linux. Para a construção deste analisador foi criado um arquivo que contém as estruturas necessárias para a execução do analisador semântico e verificação de seus respectivos erros.

A função dessa análise é a de verificar a compatibilidade dos tipos envolvidos nas construções advindas do analisador sintático. Para realizar a verificação, foram elaboradas as chamadas ações semânticas para a linguagem, as quais, dependendo da estrutura a ser analisada, permitirão verificar se os atributos pertinentes à mesma estão corretos [11].

Esta última fase da análise *front-end* é constituída por um tratador de erros que irá analisar se a compatibilidade entre os tipos envolvidos em uma determinada estrutura está de acordo com aquilo que foi definido para a linguagem. A verificação dessa compatibilidade está relacionada com os dados envolvidos em operações aritméticas, operações de comparações e chamadas a procedimentos da linguagem.

Como exemplo de verificação, cita-se a referente ao comando *CriaCentrodeServiço*. Neste caso, será analisado se os parâmetros passados ao mesmo, identificador de centro de serviço e número de servidores, são do tipo *CentrodeServiço* e número inteiro, respectivamente.

4.5 Geração de Código Intermediário

Após a fase de *front-end*, o compilador da LiSReF passa para a fase de geração de código (*back-end*). No entanto, o compilador da linguagem somente passa para esta última fase se nenhum erro for encontrado na *front-end*. Os erros que foram encontrados nesta fase são mostrados ao usuário e a geração de código não é realizada.

É importante salientar que o processador Nios II suporta somente dados do tipo inteiro e não representações numéricas em ponto flutuante. Com isso, embora a

LiSReF possua em sua estrutura dados numéricos deste último tipo, para a construção dos seus respectivos código intermediário e código objeto somente foram considerados dados do tipo inteiro.

O gerador de código intermediário para a LiSReF foi desenvolvido na linguagem C++, no qual as ações correspondentes à geração do código intermediário foram inseridas no analisador sintático, já que este se encarrega da verificação das estruturas da linguagem.

Para construir o gerador de código intermediário foi utilizado um código de três endereços, que se baseia na utilização de três endereços: dois para os operandos e um para o resultado. No esquema *x := y op x*, *x*, *y* e *z* são os operandos e *op* refere-se à algum operador, seja lógico, aritmético, booleano ou de outro tipo [1]. Esta forma de implementação visa facilitar a elaboração do código objeto para a LiSReF.

Na Tabela 2 são apresentados exemplos de construções da LiSReF e seu código intermediário correspondente.

Tabela 2. Correspondência entre trechos de código da LiSReF e o código intermediário.

Código da LiSReF	**Código Intermediário Gerado**
TempoAtual <- RecebeTempoAtual relog	_T_Atual := Tempo_Relogio
TempoFinaldaSimulação <- 1000	_T_Final := 1000
ENQUANTO (TempodoSistema < 1000) COMEÇO *comandos* FIMENQUANTO	_R1: se _T_Atual < 1000 jump _R2 jump _RProx _R2: *comandos dentro de ENQUANTO* jump _R1 _RProx: *comandos executados após o comando ENQUANTO*
EventoInicial professor (chegada)	_EvInicial := chegada
Simular chegada (Usuário, tEntreChegada, atendimento)	_Simular := chegada, tEntreChegada _usu_sistema := _usu_sistema + 1
Simular atendimento (Usuário, tAtendimento, saída)	_Simular := atendimento, tAtendimento _usu_fila := _usu_fila -1 _EstadoServidor := OCUPADO
SE (EstadoServidor = Ocupado) COMEÇO *comandos* FIMSE SENÃO *comandos* FIMSENÃO	se _Estado_Servidor = OCUPADO jump _R1 jump _RProx _R1: *comandos dentro de SE* _RProx: *comandos executados dentro de SENÃO*
SaiDoServidor cds (saída)	_SaiDoServidor := cds _usu_sistema := _usu_sistema - 1

4.6 Geração de Código Objeto

O gerador de código objeto produz o código final a ser executado. Ele depende da máquina alvo, baseando-se nas instruções que a mesma possui. Neste caso, o código a ser gerado fundamenta-se nas instruções do Nios II. O conjunto de instruções do Nios

II é relativamente pequeno e divide-se em 5 tipos: instruções de comparação, de operações aritméticas, de salto, de movimentação e de movimentação de dados [2].

Na Tabela 3 são mostrados alguns exemplos de código intermediário e sua tradução para o código objeto.

Tabela 3. Código intermediário e sua tradução para código objeto.

Código Intermediário	Código Objeto

Inicialmente
_usu_fila := 0
_usu_sistema := 0
_usu_fila := _usu_fila -1
_usu_sistema := _usu_sistema + 1
	Movhi R10, 1
	Movhi R11, 0
	Movhi R12, 0
	Sub R11, R11, R10
	Add R12, R12, R10

_R1: se _T_Atual < 1000 jump _R2 jump _RProx _R2: *comandos dentro do laço* _RProx: *comandos executados depois do laço*	_R1: Mov R13, _T_Final Movhi R14, 1000 Blt R13, R14, L2 Jmp _RProx _R2: *comandos dentro do laço*

se _EstadoServidor = OCUPADO jump _R1
jump _RProx
_R1: *comandos dentro de SE*
_RProx: *comandos executados depois de SE*
	Mov R13,_EstadoServidor
	Movhi R14, 0
	Beq R13, R14, L1
	Jmp _RProx
	_R1: *comandos dentro de SE*
	_RProx: *comandos depois de SE*

4.7 Exemplo de Programa em LiSReF

Na Figura 2 é mostrado um exemplo completo de especificação de um programa de simulação escrito em LiSReF, que simula um sistema composto por um professor e seus alunos. Neste caso, o professor atende um aluno de cada vez, ou seja, quando um estudante chega à sala do professor, se o mesmo estiver atendendo outro aluno, ele deve esperar na fila até a sua vez de ser atendido. Depois de ser atendido, o aluno deixa a sala e permite que outro aluno possa tirar suas dúvidas. Este sistema foi modelado através de uma fila M/M/1 [3], em que o professor representa o servidor, os alunos representam os usuários do serviço e a fila de espera é formada pelos alunos que aguardam atendimento.

Na primeira parte do programa, delimitada pelas palavras DECLARAÇÕES e FIMDECLARAÇÕES, encontram-se todas as variáveis a serem utilizadas, incluindo: a definição do servidor (*professor*), a criação do relógio global do sistema (*relog*), a determinação do tempo total da simulação (*TempoFinaldaSimulação*), o intervalo de tempo médio entre chegadas de alunos (*TEntreChegadas*), o tempo médio de atendimento (*TAtendimento*) e os eventos do sistema (*chegada*, *atendimento* e *saída*). Neste exemplo, o tempo de atendimento foi fixado em 10 unidades de tempo (linha 5 do programa) e o tempo entre as chegadas de clientes ao sistema foi fixado em 5 unidades de tempo (linha 4 do programa).

Após a primeira parte, tem-se o trecho relativo aos comandos que realizam a simulação, delimitado pelos termos PROGRAMASIMULAÇÃO e FIMDASIMULAÇÃO. Inicialmente, instancia-se a variável *chegada* como sendo o evento inicial da simulação e o relógio do sistema recebe o tempo atual (*TempoAtual*), com valor inicial igual à zero. Assim, a simulação é iniciada, sendo executada enquanto *TempoAtual* não ultrapassar o *TempoFinaldaSimulação*.

```
1  DECLARAÇÕES
2  CriaCentrodeServiço professor, 1;
3  CriaRelógio relog;
4  VariáveldeSimulação Inteiro TEntreChegadas <- 5;
5  VariáveldeSimulação Inteiro TAtendimento <- 10;
6  VariáveldeSimulação Inteiro chegada <- 1;
7  VariáveldeSimulação Inteiro atendimento <- 2;
8  VariáveldeSimulação Inteiro saída <- 3;
9  TempoFinaldaSimulação <- 10000;
10 FIMDECLARAÇÕES
11
12 PROGRAMASIMULAÇÃO
13 TempoAtual <- RecebeTempoAtual(relog);
14 EventoInicial professor(chegada);
15
16 ENQUANTO(TempoAtual <= TempoFinaldaSimulação)
17 COMEÇO
18 Simular chegada(Usuário,TEntreChegadas,atendimento);
19 EstadoServidor <- RecebeEstadoServidor (professor);
20   SE(EstadoServidor = Desocupado)
21   COMEÇO
22   Simular atendimento(Usuário, TAtendimento, saída);
23   Simular saída(Usuário, TempodoSistema, saída);
24   SaiDoServidor professor(saída);
25   FIMSE
26
27 FIMENQUANTO
28 FIMDASIMULAÇÃO
```

Figura 2. Exemplo da especificação de uma simulação em LiSReF [11].

6 Conclusões

Este trabalho apresentou uma linguagem para simulação de redes de fila, baseada no uso de construções algorítmicas, que auxiliam o programador na implementação de programas de simulação para sistemas modelados com redes de fila. A linguagem apresenta as construções básicas de comandos de repetição, atribuição, comando condicional, tipos e variáveis, além de toda a estrutura necessária para a

implementação de um programa de simulação. Essa estrutura envolve, entre outros, o relógio da simulação, a lista de eventos futuros e o tratamento estatístico dos resultados obtidos.

A grande vantagem da LiSReF está relacionada com a facilidade oferecida ao programador. Através de seus comandos e estruturas redigidos no português estruturado e de sua forma algorítmica, a linguagem facilita e simplifica a redação e o entendimento de um ambiente de simulação. Além disso, as mensagens de erro geradas pelo compilador da linguagem são de fácil entendimento, o que possibilita ao usuário a correção dos erros do programa sem grande dificuldade.

Os próximos passos do trabalho envolvem modificações na implementação dos modelos de redes de filas e a inclusão de funções de distribuições de probabilidade. Além disso, para que seja possível gerar código compatível para arquiteturas x86, deve-se alterar o *back-end* do compilador da LiSReF. Pretende-se também, utilizar a linguagem algorítmica para o ensino de simulação e conceitos de modelagem de sistemas usando filas. Posteriormente, poderá ser efetuada uma avaliação comparativa do aprendizado com o uso da linguagem algorítmica e com o uso de uma linguagem de simulação.

Agradecimentos. As autoras expressam seus agradecimentos à Fundunesp pelo apoio.

Referências

1. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Massachusetts (2007).
2. Altera, www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.
3. Banks, J., Carson, J. S., Nicol, D. M., Nelson, B. L.: Discrete-Event System Simulation. Prentice Hall, New Jersey (2004).
4. Di Chiacchio, R. L. C.: Biblioteca Orientada a Objetos para Simulação de Redes de Fila. Projeto final de curso. Monografia de Projeto Final de Curso de Graduação em Ciência da Computação – Universidade Estadual Paulista, São José do Rio Preto, SP (2005).
5. Fortier, P. J.; Michel, H. E.: Computer Systems Performance Evaluation and Prediction. Digital Press, Burlington (2003).
6. Freitas, P. J.: Introdução à Modelagem e Simulação de Sistemas. Visual Books Ltda, Florianópolis (2001).
7. GNU Operating Systems, www.gnu.org/software/bison/manual/pdf/bison.pdf.
8. Levine, J. R., Mason, T., Brown, D.: Lex & Yacc. O'Reilly, Beijing (1992).
9. Morell, L. J.: Algorithms in Genesis. Journal of Computing Sciences in Colleges. 20, 48--54 (2005).
10. Muntha, S.; Morell, L. J.: Adding Object Orientation to Genesis. Journal of Computing Sciences in Colleges. 21, 101--106 (2006).
11. Oliveira, V. G.: Uma Linguagem Algorítmica para Simulação de Redes de Filas. Projeto final de curso. Monografia de Projeto Final de Curso de Graduação em Ciência da Computação – Universidade Estadual Paulista, São José do Rio Preto, SP (2006).
12. Rice, S. V.; Marjanski, A.; Markowitz, H. M.; Bailey, S. M.: The Simscript Programming Language for Modular Object-Oriented Simulation. In: Proceedings of the 37th Winter Simulation Conference. pp. 621--630. Orlando, Florida (2005).
13. Schildt, H.: C Completo e Total. Makron Books, São Paulo (1997).