UNIVERSIDADE DE SÃO PAULO

Desenvolvimento de um gerador de aplicação para simulação de sistemas discretos

ROBERTA SPOLON

RENATA SPOLON

MARCOS JOSÉ SANTANA

REGINA H. C. SANTANA

Nº 11

NOTAS

Instituto de Ciências Matemáticas de São Carlos

Instituto de Ciências Matemáticas de São Carlos

ISSN - 0103-2577

Desenvolvimento de um gerador de aplicação para simulação de sistemas discretos

ROBERTA SPOLON

RENATA SPOLON

MARCOS JOSÉ SANTANA

REGINA H. C. SANTANA

Nº 11

N O T A S D O I C M S C

Série Computação

São Carlos
jul. / 1994

Desenvolvimento de um Gerador de Aplicação para Simulação de Sistemas Discretos

Roberta Spolon

Renata Spolon

Marcos José Santana

Regina H. C. Santana

Grupo de Programação Concorrente e Sistemas Distribuídos
Departamento de Computação e Estatística
Instituto de Ciências Matemáticas de São Carlos
Universidade de São Paulo

São Carlos
julho de 1994

Sumário

Lista de Figuras
Resumo

1. Introdução 01

2. Conceitos Básicos: Simulação e Geradores de Aplicação 03
  2.1 Simulação 03
    2.1.1 Classificação dos Modelos de Simulação 03
    2.1.2 Desenvolvimento do Modelo e Testes 05
    2.1.3 Linguagens de Simulação 06
  2.2 Geradores de Aplicação 06
    2.2.1 Componentes de um Gerador de Aplicação 06
    2.2.2 Vantagens na Utilização de Geradores de Aplicação 09

3. Implementação do Gerador de Aplicação para Simulação 11
  3.1 Estrutura Geral do Ambiente de Simulação Automático 11
  3.2 As Fases no Processo de Execução do Gerador 12
    3.2.1 Modelagem: Definição do Modelo e Parâmetros 12
    3.2.2 Formação das Tabelas de Especificações 13
  3.3 Módulos que Compõem o Gerador 14
    3.3.1 Módulo INTERF.C 14
    3.3.2 Módulo DEF.H 15
    3.3.3 Módulo PROTOT.H 15
    3.3.4 Módulo PRINCIPAL 15
    3.3.5 Módulo UTIL.C 18
    3.3.6 Módulo DIST_TAB.C 18
    3.3.7 Módulo REESC.C 19

3.4 A Geração do Código 19
3.4.1 A Descrição de Produtos 20
3.4.2 Processo de Execução do Gerador 22
3.4.3 A Execução 24
4. Exemplos 30
4.1 Exemlo 1 30
4.2 Exemplo 2 32
4.3 Exemplo 3 34
4.4 Exemplo 4 36
4.5 Exemplo 5 39
5. Considerações Finais 42
6. Referências Bibliográficas 43
7. Bibliografia 45

Lista de Figuras

Figura 2.1: Relação entre Atividade, Processo e Evento 04

Figura 2.2: Processo de Modelagem e Análise 05

Figura 2.3: Relacionamento entre os Módulos de um Gerador 07

Figura 2.4: Processo de Desenvolvimento com um Gerador 09

Figura 3.1: Visão Geral do ASiA 12

Figura 3.2: Modelo com Único Recurso 13

Figura 3.3: Extensão do Modelo de Redes de Filas 18

Figura 3.4: Gabarito 21

Figura 3.5: As Fases no Processo de Execução do Gerador 22

Figura 3.6: Fluxo de Dados no Gerador 25

Figura 4.1: Exemplo 1 30

Figura 4.2: Exemplo 2 32

Figura 4.3: Exemplo 3 34

Figura 4.4: Exemplo 4 37

Figura 4.5: Exemplo 5 39

Resumo

Este trabalho apresenta um estudo sobre simulação e geradores de aplicação, ressaltando suas características, classificação e exemplos de aplicação. Discute-se os aspectos gerais da implementação e execução de um gerador de aplicação para simulação de sistemas discretos e exemplos de utilização do mesmo.

O gerador faz parte de um ambiente de simulação denominado ASiA (Ambiente de Simulação Automático), em desenvolvimento no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMSC-USP.

Com a utilização do gerador é possível que usuários com pouca experiência em programação e mesmo profissionais tenham condições de elaborar uma simulação, minimizando seu trabalho e aumentando a qualidade do código produzido.

1. Introdução

O desenvolvimento de um programa de simulação pode consistir uma tarefa árdua, exigindo conhecimentos não só sobre o problema em estudo, como também das técnicas de modelagem e análise. Nesse sentido, e considerando as vantagens associadas à utilização de geradores de aplicação (especificadas no item 3.2), foi desenvolvido um Gerador de Aplicação para simulação de sistemas discretos. Com a utilização do Gerador é possível que usuários com pouca experiência em programação tenham condições de elaborar uma simulação, além de minimizar o trabalho dos profissionais que possuam o conhecimento necessário para implementar um programa de simulação.

O objetivo principal deste relatório técnico é detalhar a construção desse gerador e apresentar alguns exemplos da sua utilização.

No decorrer do trabalho o gerador desenvolvido será referenciado como Gerador (Gerador de Aplicação). Ressalta-se que o Gerador faz parte de um ambiente de simulação denominado ASiA (Ambiente de Simulação Automático), em desenvolvimento no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMSC-USP.

Com o ASiA, o usuário do sistema de simulação não precisa se preocupar com a transcrição do modelo em um programa, necessitando apenas especificar, através do editor gráfico, o modelo do sistema em questão e os parâmetros necessários à implementação do programa de simulação. Se o modelo apresenta algumas particularidades (como tipos específicos de estatísticas), o usuário pode especificar o modelo e depois incluir no programa gerado as particularidades requeridas via editor de textos.

O Gerador foi desenvolvido no LaSD (Laboratório de Sistemas Digitais) do ICMSC-USP, utilizando como plataforma máquinas PC compatíveis com IBM. A linguagem de programação adotada foi C, utilizando-se o compilador Borland C++ versão 3.1 e o sistema operacional MS-DOS versão 5.0.

Este trabalho está organizado da seguinte forma:

. Na seção 2 são apresentados conceitos básicos sobre simulação e geradores de aplicação,

. A seção 3 faz uma descrição geral do ambiente ASiA e do gerador desenvolvido, apresentando considerações referentes à implementação do mesmo.

. Na seção 4 são apresentados exemplos de aplicação do gerador.

. A seção 5 encerra este relatório técnico apresentando uma conclusão geral do trabalho.

2. Conceitos Básicos: Simulação e Geradores de Aplicação

2.1 Simulação

Simulação é uma experiência ou ensaio realizado com o auxílio de modelos, possibilitando a representação das características do comportamento de um sistema físico ou lógico. Quatro objetivos principais justificam a sua utilização: como uma ferramenta explanatória para a definição de um problema, para a detecção de elementos críticos, para a análise e avaliação de soluções propostas e para o planejamento de desenvolvimentos futuros [MAR80].

2.1.1 Classificação dos Modelos de Simulação

Modelos de simulação são classificados em discretos e contínuos, dependendo de como ocorrem as alterações no seu estado [MAC87,SOA90].

Modelos Discretos: representam sistemas onde as mudanças no seu estado ocorrem em pontos específicos e descontínuos do tempo simulado, alterando o estado do sistema espontaneamente. Nesse tipo de modelo normalmente há disputa por recursos escassos e filas onde os elementos do sistema esperam pela liberação do recurso solicitado. Os atrasos entre mudanças de estado são determinados estatisticamente, com o intervalo selecionado de acordo com alguma distribuição amostral [SHA88].

Modelos Contínuos: são modelos nos quais as variações de estado podem ocorrer continuamente ao longo do tempo de simulação. Normalmente são descritos através de equações diferenciais.

A composição dinâmica de um sistema discreto é definida em termos de entidades denominadas atividades, processos e eventos [MAC87, TAN94].

Atividade: menor unidade de trabalho no sistema, porém dependente da visão que se tenha do mesmo.

Processo: um conjunto de atividades logicamente relacionadas constitui um processo [MAC75, MAC87].

Evento: corresponde a uma mudança de estado de alguma entidade do sistema. O término de uma atividade é um evento cuja ocorrência pode iniciar outras atividades subseqüentes. Os eventos controlam a seqüência de atividades dentro de um processo.

Em simulação discreta, atividades, eventos e processos são construções utilizadas para descrever o seu comportamento dinâmico, servindo como base para as linguagens de simulação. Um sistema é visto dinamicamente como uma coleção de processos interagindo, com essas interações controladas e coordenadas pela ocorrência de eventos [MAC75].

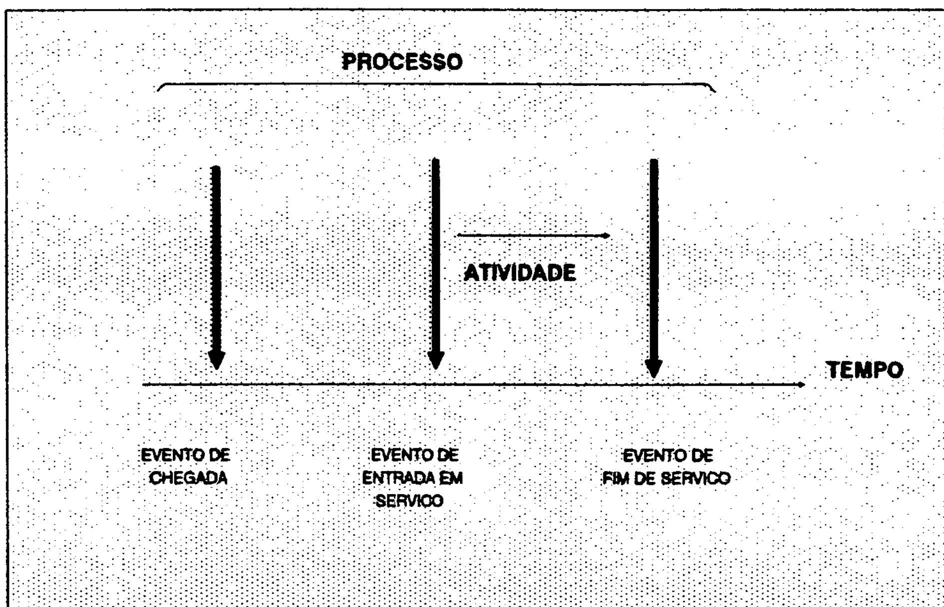

Figura 2.1: Relação entre Atividade, Processo e Evento

Desse modo, a formulação de um modelo para simulação discreta pode ser realizada de três formas [SOA90]:

- pela descrição das atividades nas quais as entidades do sistema se envolvem, denominada simulação orientada a atividades;

- pela descrição dos processos através dos quais as entidades do sistema fluem, denominada simulação orientada a processos e

- pela definição das mudanças nos estados que podem ocorrer em cada tempo de evento, denominada simulação orientada a eventos.

2.1.2 Desenvolvimento do Modelo e Testes

O processo de modelagem e testes no desenvolvimento de uma simulação é constituído por diversas fases, conforme a figura 2.2. O processo é dividido em três fases principais: desenvolvimento, testes e análise. Ressalta-se que a seqüência linear mostrada na figura raramente é seguida na prática, ocorrendo muitas interações entre as diversas fases apresentadas [MAC87, TAN94].

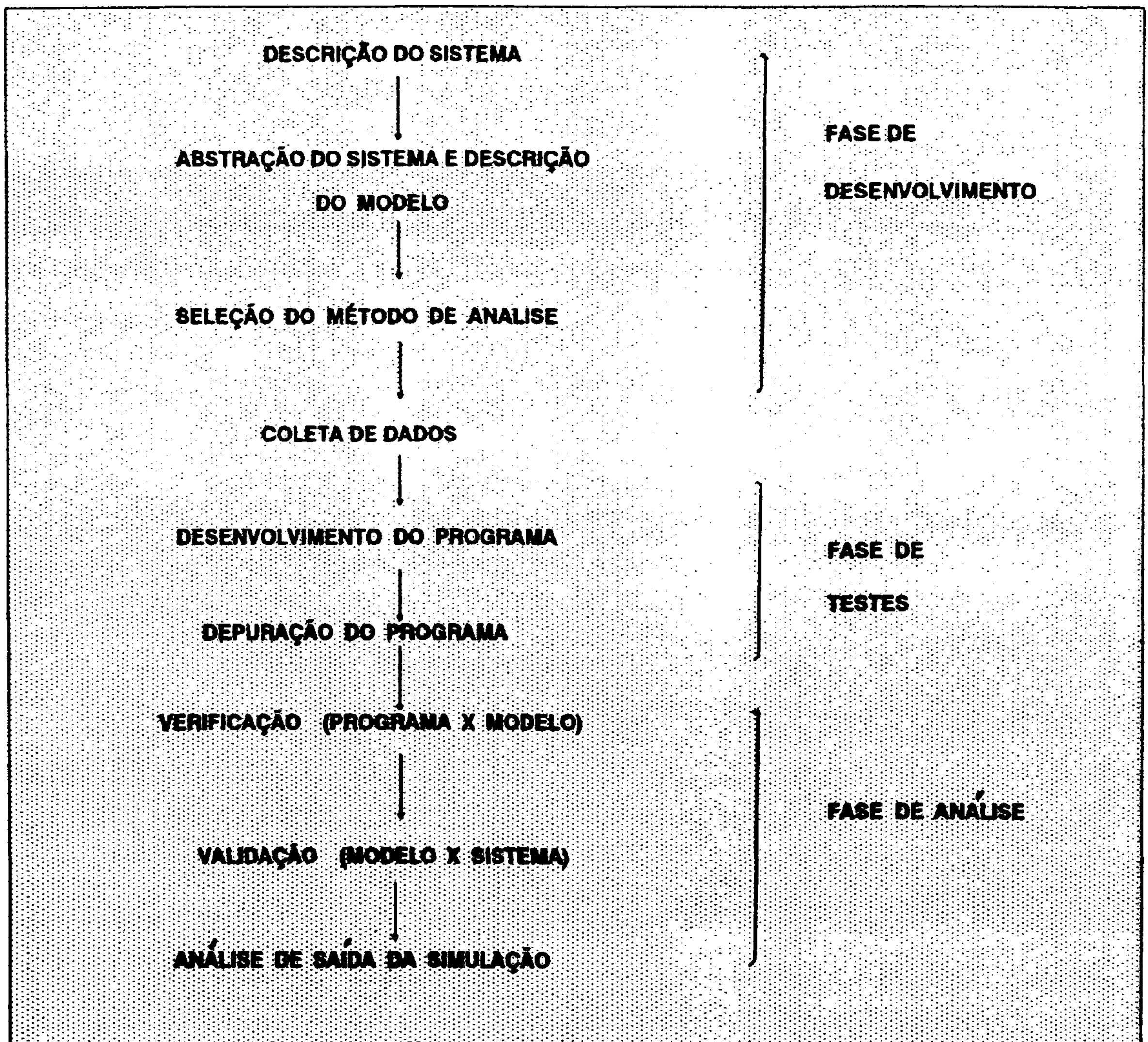

Figura 2.2: Processo de Modelagem e Análise

2.1.3 Linguagens de Simulação

O uso da simulação como uma ferramenta para a análise de sistemas deu origem a uma série de linguagens especificamente projetadas para esse fim. Essas linguagens impõem uma certa estruturação nos modelos e simplificam suas soluções. Ressalta- se que alguns pacotes como SMPL, CSIM e SIMTOOLS são classificados como linguagens na literatura [MAC75, MAC87, SCH86], porém não constituem exatamente linguagens, mas sim extensões funcionais (bibliotecas) que permitem o desenvolvimento de programas de simulação em linguagens convencionais, oferecendo todos os recursos e facilidades disponíveis na linguagem hospedeira. Nos exemplos de linguagens citados incluem-se essas extensões.

Exemplos:

- Linguagens orientadas a eventos: GASP, SIMSCRIPT, SMPL [MAC75, MAC87], SLAM, SIMAN [SHA88].

- Linguagens orientadas a processos: ASPOL, SIMULA, GPSS [MAC75], HPSIM [SHA88] e CSIM [SCH86].

2.2 Geradores de Aplicação

Há algum tempo os custos de desenvolvimento de "software" ultrapassaram o custo relativo do "hardware", tornando cada vez mais premente e importante a busca de soluções para esse problema. Como alternativa tem-se o desenvolvimento de geradores de aplicação, ferramentas que possibilitam um grande aumento de produtividade, além de melhorar a qualidade dos sistemas produzidos [MAS93, MEI91a].

2.2.1 Componentes de um Gerador de Aplicação

Geradores de aplicação são ferramentas que, a partir de uma especificação em alto nível de um problema implementável, transformam automaticamente essa

especificação na implementação do problema [MEI91a]. Para alterar o produto final (implementação), basta modificar a especificação e executar novamente o gerador.

Normalmente, os geradores de aplicação são constituídos por três módulos principais: um módulo de interface, um módulo analisador de especificações e um módulo gerador de produtos [LUK86]. A figura 2.3 mostra o relacionamento entre os módulos.

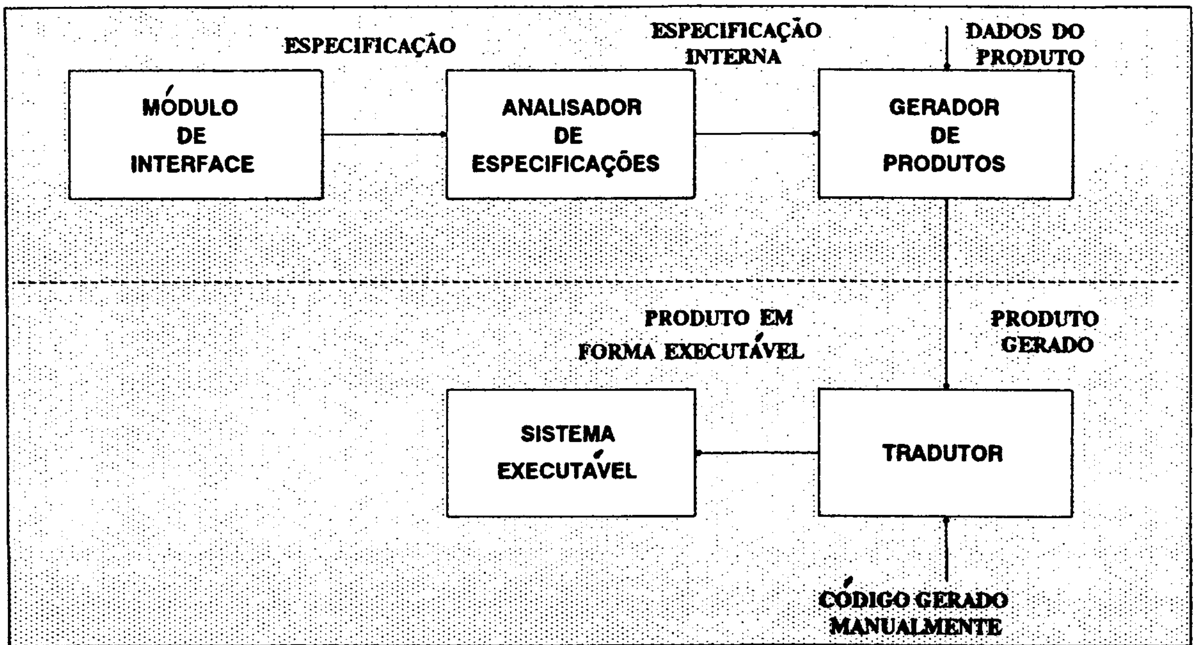

Figura 2.3: Relacionamento entre os Módulos de um Gerador

Descrição dos Módulos

. **Módulo de Interface: responsável pela interface com o usuário, com o objetivo de obter uma especificação consistente e completa do problema.**

Nem sempre esse módulo está presente nos geradores e neste caso, a entrada dos dados é textual. A interface pode utilizar também formas gráficas ou um diálogo interativo onde o usuário seleciona as opções através de um menu.

. Módulo Analisador de Especificações: responsável pelas análises sintática e semântica dos dados de entrada, para produzir as estruturas de dados intermediárias utilizadas pelo módulo Gerador de Produtos. As estruturas são dependente do tipo de problema e podem ser qualquer combinação de tabelas, árvores sintáticas de derivação e outras estruturas de dados [MEI91a].

. Módulo Gerador de Produtos: encarregado de gerar o produto desejado pelo usuário, além de uma documentação (se necessário) para auxiliar o usuário na compreensão e execução do programa.

O processo básico para desenvolver um sistema com o auxílio de um gerador começa com a especificação, fornecida pelo usuário ao gerador através da interface. O gerador então cria o produto de aplicação na linguagem de programação alvo. Finalmente o produto, juntamente com o código produzido manualmente (se necessário) é compilado para produzir o sistema executável. A figura 2.4 demonstra esse processo.

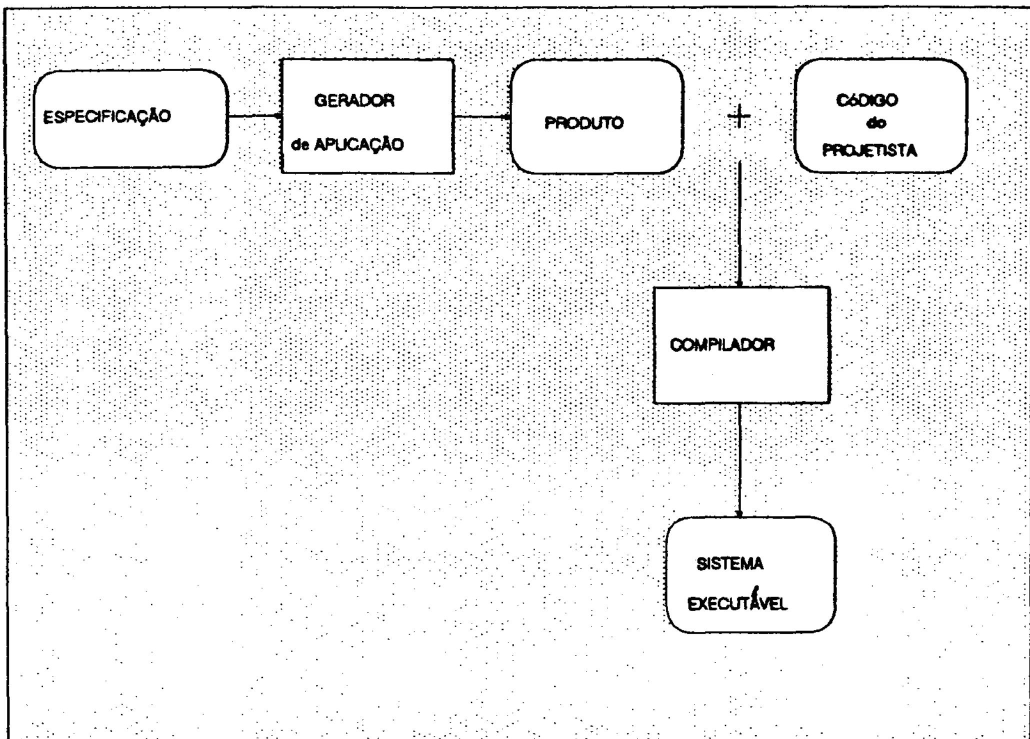

Figura 2.4: Processo de Desenvolvimento com um Gerador

Exemplos de Aplicações: [SPO93a]

- Extração em Banco de Dados: TTU e IMX;
- Protocolos de Comunicação: Compilador Estelle-C
- Aplicações com interface gráfica: TAGS e MICROSTEP
- Simulação: PASSIM e Draft.

2.2.2 Vantagens na utilização de Geradores de Aplicação

A utilização de geradores de aplicação para automatizar o processo de implementação de "softwaares" apresenta muitas vantagens. Além do aumento de produtividade na fase de desenvolvimento, não ocorrem erros na tradução da especificação para a implementação, uma vez que essa atividade é realizada automaticamente pelo gerador. A tarefa de manutenção é facilitada devido à uniformidade e modularidade do código produzido [MEI91a].

A especificação é mais fácil de ser escrita, lida e implementada, por não conter detalhes de especificação. Para corrigir um erro de especificação não é necessário alterar o código produzido, bastando alterar a especificação de entrada. Além disso, os parâmetros são especificados apenas uma vez para a geração do código.

Com o auxílio de geradores de aplicação, as aplicações podem ser geradas e mantidas por pessoas que não tenham conhecimento sobre linguagens de programação [MEI91a].

Apesar dos geradores serem restritos para a criação de produtos específicos para a sua área de aplicação e na prática gerarem apenas parte de um sistema, depois de prontos estes confirmam as suas principais vantagens: redução no custo de desenvolvimento, aumento da produtividade e boa qualidade no código produzido.

3. Implementação do Gerador de Aplicação para Simulação

O desenvolvimento do Gerador visou satisfazer alguns conceitos fundamentais de Engenharia de Software [PRE87]:

- modularidade: o sistema foi implementado em módulos que são integrados para satisfazer os requisitos do sistema;

- portabilidade: o "software" foi desenvolvido o mais independente possível do sistema operacional utilizado, visando uma maior portabilidade;

- manutenabilidade: durante a elaboração do sistema, algumas decisões foram tomadas para permitir uma melhor manutenção no decorrer do tempo de vida útil do sistema, como uma completa documentação do mesmo.

3.1 Estrutura Geral do Ambiente de Simulação Automático

O Gerador faz parte de um ambiente de simulação denominado Ambiente de Simulação Automático (ASiA), cujo objetivo é afastar o usuário do sistema da tarefa de transcrição do modelo em um programa de simulação. O usuário irá fornecer o modelo do sistema considerado e os parâmetros necessários, sendo as demais tarefas executadas automaticamente pelo Ambiente de Simulação. A figura 3.1 apresenta uma visão geral do ASiA.

O gerador desenvolvido implementa então o módulo Gerador de Produtos, criando programas de simulação em SMPL (uma extensão funcional da linguagem C para simulação orientada a eventos [MAC87]).

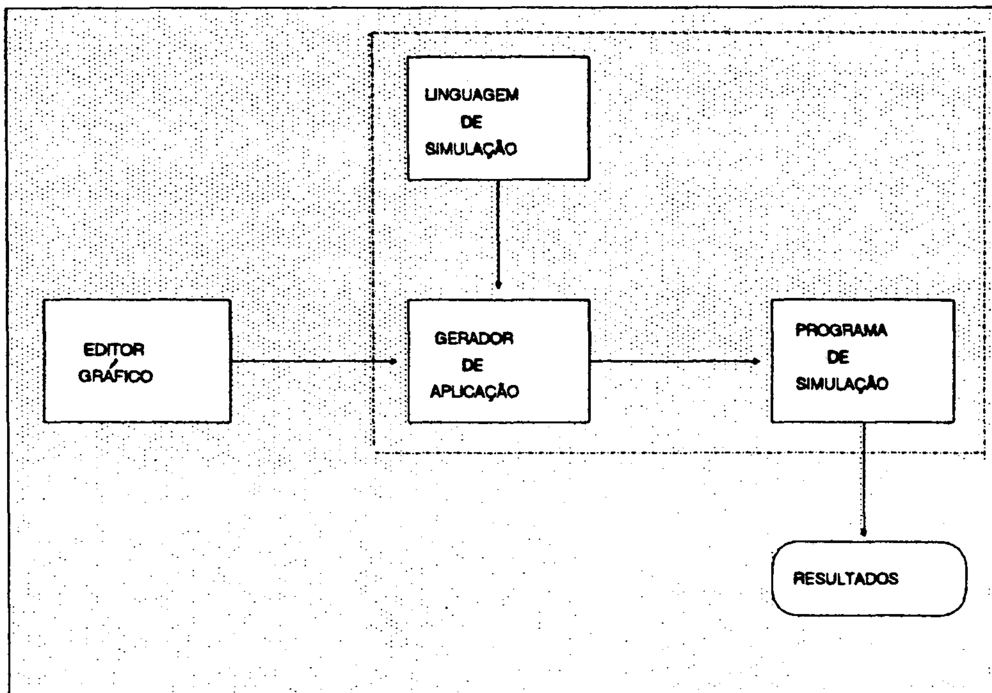

Figura 3.1: Visão Geral do ASiA

3.2 As Fases no Processo de Execução do Gerador

A execução do ASiA está dividida em 4 estágios distintos: modelagem, preparação das tabelas de especificações, geração do código e estágio de saída. Os três primeiros estágios encontram-se em desenvolvimento e a geração de código será discutida no item 3.4.

3.2.1 Modelagem: Definição do Modelo e Parâmetros

No ASiA, o editor gráfico é responsável pela interface com o usuário. A especificação do modelo (basicamente um modelo de redes de filas [SOA90]) é feita através de manipulação direta na tela e de menus, que oferecem ao usuário a possibilidade de selecionar blocos básicos para compor o modelo final. Dentre os blocos básicos disponíveis têm-se por exemplo recursos com uma fila ou recursos com várias filas.

Definido o modelo, o usuário parametriza o mesmo através de diversas opções oferecidas por um menu [SPO93b]. Como exemplo, considerando-se o bloco básico representado na figura 3.2, os seguintes parâmetros podem ser fornecidos:

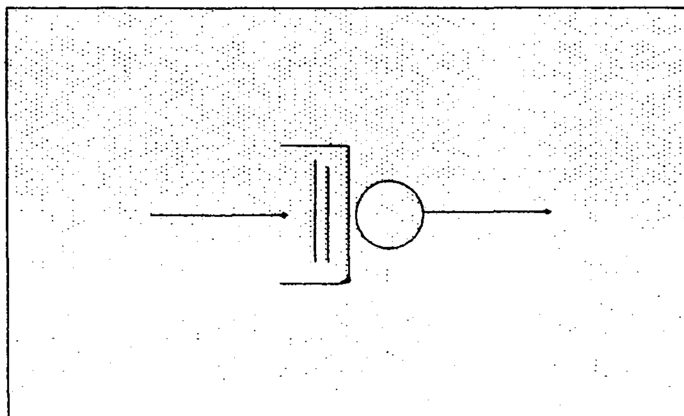

Figura 3.2: Modelo com Único Recurso

. tempo médio entre chegadas, distribuição entre chegadas, número de servidores, tempo médio de serviço, distribuição para o tempo de serviço, escolha da seqüência para geração de números aleatórios, "trace", "batch means", entre outros.

3.2.2 Formação das Tabelas de Especificações

Escolhido o modelo e determinados os parâmetros necessários, todas essas informações são estruturadas em tabelas, as quais servirão como entrada para o Gerador. Essa atividade, juntamente com as análises sintática e semântica das informações fornecidas, são realizadas pelo Editor Gráfico. São utilizadas sete tabelas, descritas a seguir:

- Modelo: contém informações genéricas do modelo a ser implementado;

- Recursos: contém informações particulares sobre cada um dos recursos que compõem o modelo. Como essa tabela constitui uma lista encadeada, será referenciada como lista;

- Reesc: tabelas de tempos para reescalonamento de eventos, para o caso de recursos sem fila;

- Ciclo: possibilita ao usuário a definição de ciclos ("loop's") no modelo;

- Distrib_Tab: permite que o usuário defina novas distribuições de probabilidade (através das informações contidas nessa tabela, a implementação da distribuição é gerada (ver item 3.3.6));

- Entradas: possibilita mais de uma entrada no sistema;

- Fechado: identifica os eventos que devem ser escalonados antes do início da simulação, sendo utilizada para modelos do tipo fechado.

3.3 Módulos que Compõem o Gerador

O Gerador é constituído pelos módulos INTERF.C, DEF.H, PROTOT.H, PRINCIPAL.C, UTIL.C, DIST_TAB.C e REESC.C, descritos a seguir.

3.3.1 Módulo INTERF.C

Como a atividade de desenvolvimento dos módulos que compõem o ASiA (incluindo o Gerador) está sendo executada paralelamente, o Gerador apresenta uma interface simples e própria para a criação das diversas tabelas necessárias para a realização de testes.

A interface desenvolvida é textual, sendo composta por diversos módulos, e o usuário se limita a responder questões do tipo Sim ou Não ou a fornecer os parâmetros especificados. Não é feita análise sintática ou semântica dos dados fornecidos, já que essas análises são realizadas pelo Editor Gráfico e, quando o ambiente ASiA for integrado (módulos Gerador e Editor Gráfico), esta interface de testes será retirada do sistema.

Os seguintes módulos constituem a interface, para a criação das tabelas de especificação do modelo a ser implementado:

- criamod.c: contém as primitivas necessárias para possibilitar que o usuário especifique as principais características do modelo a ser implementado;
- criarec.c.c: permite ao usuário especificar cada um dos recursos que compõem o modelo, definindo suas características particulares;
- criaent.c: possibilita a definição de várias entradas no modelo;
- criatab.c: permite ao usuário a definição de novas distribuições de probabilidade;
- criaf.c: possibilita a definição dos recursos para os quais um determinado evento deva ser escalonado antes do início da simulação;
- criarees.c: se o usuário definiu recursos sem fila no seu modelo, este módulo permite a especificação de uma tabela contendo os tempos para uma nova tentativa de utilização do recurso caso encontre este ocupado.

3.3.2 Módulo DEF.H

Contém as definições das estruturas de dados utilizadas pelo Gerador, além de variáveis e constantes globais necessárias durante a execução do sistema. Esse módulo é incluído no módulo PRINCIPAL.C.

3.3.3 Módulo PROTOT.H

Contém os protótipos de todas as primitivas definidas nos módulos Principal, Util, Reesc e Dist_Tab. Esse módulo é incluído no módulo PRINCIPAL.C.

3.3.4 Módulo PRINCIPAL.C

Implementa o núcleo do Gerador, ou seja, define as principais primitivas utilizadas no processo de geração de código. Cada uma das primitivas é responsável por gerar um trecho específico de código no produto final (programa de simulação).

Esse módulo inclui primitivas para:

. Gerar os principais comandos do SMPL, responsáveis pelas seguintes atividades:

. preparação do modelo;

. definição e controle dos recursos;

. escalonamento e geração de eventos;

. geração de variáveis aleatórias;

. depuração do código e

. coleta de dados e emissão de relatórios.

. Utilização de novos relatórios:

Além de possibilitar ao usuário a utilização do relatório padrão do SMPL (com informações sobre o tempo de execução da simulação, utilização dos recursos, período médio ocupado, comprimento médio da filas associadas a cada recurso e contadores de operações realizadas), o Gerador implementa outros relatórios com as seguintes informações:

. estatísticas sobre o número máximo e mínimo de clientes nas filas:

. porcentagem de vezes em que o cliente encontra uma determinada fila vazia:

. Distribuição de Probabilidade Triangular

O SMPL implementa as seguintes distribuições: exponencial, erlang, normal e hiper-exponencial. Porém, a distribuição triangular pode ser adequada em situações onde se conhece o intervalo de variação de uma variável aleatória, além de seu valor mais provável. Um exemplo de aplicação é a distribuição do tempo de busca de informações em um disco, onde tem-se os tempos máximos e mínimos da busc e o tempo de busca de ocorrência mais freqüente. O Gerador então implementa esta distribuição aplicando o método da Transformada Inversa [SOA90], e o código resultante foi acoplado ao SMPL.

. Representar a posse simultânea de recursos

A posse simultânea de recursos em redes de filas ocorre por exemplo em modelos de sistemas de "I/O", nos quais um cliente requer o serviço de um disco, além do serviço adicional de um canal para a transferência dos dados, ou mesmo a utilização de uma "CPU" por uma tarefa, após ter alocado memória suficiente para a realização da tarefa. Porém, quando se representa um modelo através de redes de filas convencionais, essa posse simultânea de recursos não pode ser representada [SOA90].

Chandy [CHA78], Jacobson [Jac82] e Soares [SOA90] sugerem uma extensão poderosa e conseqüente aproximação para a resolução analítica de modelos de redes de filas, possibilitando a representação de características complexas que existem nos sistemas reais. Essa representação utiliza os símbolos da ferramenta RESQ (Research Queueing Package) empregada na modelagem e simulação discreta orientada a processos, sendo denominada estrutura de rede de filas estendida [SOA90].

Nessa representação estendida existem os recursos passivos, os quais são utilizados para modelar um recurso com número limitado de elementos. Outra característica é que esses recursos são alocados a um cliente enquanto o mesmo cliente recebe serviços em outros recursos, e então liberados em algum outro ponto do modelo. Nos recursos não existem servidores realizando serviços, mas apenas representam um número finito de elementos de um recurso, tal como "buffers", unidades de memória, canais de comunicação, etc. Os recursos que oferecem algum tipo de serviço ao(s) cliente(s) são denominados recursos ativos.

A representação sugerida para o sincronismo na posse simultânea de recursos é demonstrada na figura 3.3. O símbolo 1 indica a alocação do recurso ao cliente, enquanto o símbolo 2 representa a sua liberação.

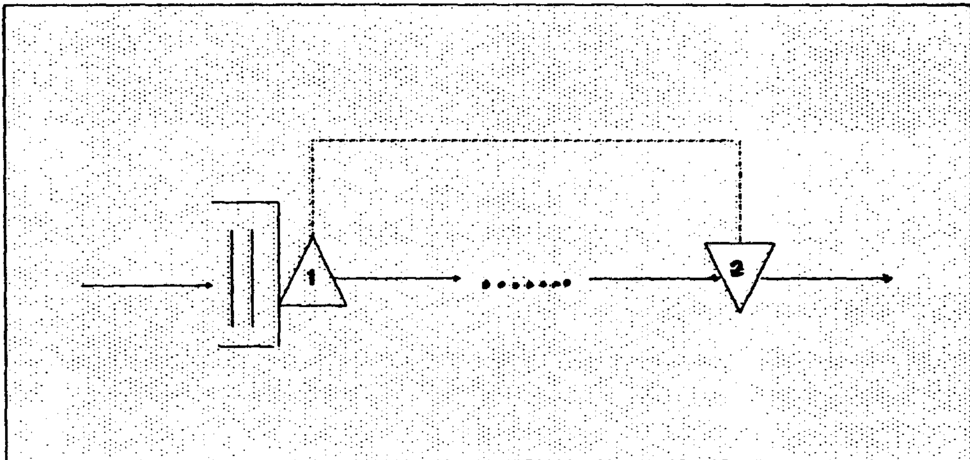

Figura 3.3: Extensão do Modelo de Rede de Filas

3.3.5 Módulo UTIL.C

Contém as primitivas (ferramentas) comuns utilizadas pelos diversos módulos do sistema para abertura/fechamento de arquivos, leitura/gravação de registros, manipulação de "strings", mensagens de erros e auxílio na geração de eventos específicos. Algumas das primitivas para a manipulação de "strings" foram implementadas para dar maior eficiência à execução do Gerador.

3.3.6 Módulo DIST_TAB.C

O exercício da simulação requer a geração de amostras aleatórias para indicar, por exemplo, o tempo da próxima chegada de um cliente ao sistema ou tempo de serviço em um recurso determinado. Pode-se obter tais amostras a partir de uma distribuição teórica (se os dados se ajustam a uma determinada distribuição) ou diretamente das tabelas fornecidas pelo usuário.

Além das distribuições de probabilidade pré-definidas pelo SMPL (exponencial, erlang, hiperexponencial e normal), o ambiente ASiA possibilita ao usuário a determinação de novas distribuições, através de tabelas fornecidas ao durante o processo de especificação do modelo. O usuário fornece ao Editor Gráfico os valores e números de ocorrência observados e este produz a distribuição de freqüência

correspondente. A partir dessa distribuição o Gerador obtém a distribuição de freqüência acumulada, armazenada no arquivo DTAB.C. O processo para a obtenção de amostras aleatórias é definido em [MAC87].

O Gerador limita o número de novas distribuições (no máximo 10) pois é necessário definir uma estrutura de dados estática (módulo DEF.H) para armazenar a identificação dos arquivos utilizados. Esta estrutura pode ser ampliada, caso seja necessário.

O módulo Dist_Tab contém então as primitivas necessárias para a implementação das distribuições de freqüência acumulada. As distribuições são armazenadas no arquivo DTAB.C, a ser incluído no produto final (programa de simulação).

3.3.7 Módulo REESC.C

Contém as primitivas necessárias para a geração da tabela de tempos e comandos necessários para a sua utilização. Essa tabela é empregada no reescalonamento de eventos para recursos sem fila.

Para o SMPL, todo recurso tem uma fila associada ao mesmo, e um número variado de servidores. Entretanto, existem modelos cujos recursos podem não ter fila, exigindo um tratamento diferenciado dos demais.

Nesses recursos, quando um cliente chega e não há servidores disponíveis para atendê-lo, deve-se definir uma política para tentar novamente a utilização do recurso. Essa política, definida como política de reescalonamento, é determinada pelo usuário através de uma tabela onde são especificados os tempos para escalonar uma nova tentativa de utilização do recurso.

3.4 A Geração do Código

O Gerador utiliza no processo de geração do código a especificação do modelo em questão (já estruturadas nas diversas tabelas relatadas no item 3.2.2), além de uma descrição de produtos.

3.4.1 A Descrição de Produtos

A descrição de produtos é um tipo de gabarito, parecido com o produto final [MEI91a], onde trechos de códigos na linguagem alvo (SMPL) estão misturados com comandos que indicam ao Gerador como buscar as informações necessárias nas várias tabelas que serão utilizadas. Os trechos de código são copiados para o produto final, enquanto os demais comandos (precedidos por um caracter especial), determinam quais as primitivas do Gerador a serem executadas. A descrição utilizada será referenciada como Gabarito.

A estrutura do Gabarito reflete exatamente o produto final a ser gerado, no caso, um programa de simulação em SMPL. Este Gabarito é específico para o tipo de produto a ser gerado, juntamente com o núcleo do gerador, enquanto o Editor Gráfico é genérico e independente do tipo de linguagem ou sistema de simulação adotado. A figura 3.4 apresenta o Gabarito utilizado pelo Gerador.

```
#include "stdio.h"
#include "conio.h "

%Idistribuicao_tabela

main()
{
%0define_tempo_num_max
 int Event = 1, Customer = 1, Aleatorio;
%Agera_contadores_loop
%1define_variaveis

%2define_smpl
%Hmodulo_estat
%3define_server
%4escalona_primeiro_evento
%Bgera_trace

%5gera_loop
{
%Jwarm_up
%6gera_cause
%7gera_switch
{
%8gera_eventos
}

%9gera_relatorio_padrao
%Cgera_relatorio_estatisticas_max_min
%Dgera_relatorio_estatisticas_fila_vazia
}
```

Figura 3.4: Gabarito

3.4.2 Processo de Execução do Gerador

A geração de código no Gerador é vista como uma atividade independente, sendo realizada em 3 fases distintas sobre o produto final (no caso, o programa de simulação em SMPL):

. definições;

. preparação do modelo e

. implementação.

A figura 3.5 ilustra essas fases em um programa de simulação em SMPL para uma fila M/M/1 (um sistema com uma única fila, um único servidor, população infinita de clientes, tempo entre as chegadas de clientes e tempo de serviço exponencialmente distribuídos). O estudo sobre teoria de filas está fora do objetivo deste trabalho, para maiores detalhes consultar Soares [SOA90].

```
#include "smpl.h"
main()
{
 /* Definições */
 real T_chegada=200.00,     /* Tempo médio de chegada de um cliente */
      T_servico=100.00,     /* Tempo médio de servico de um cliente */
      T_execucao=200000.0;  /* Tempo de simulação */
 int cliente=1,evento, servidor;
 /*  Preparação do modelo  */
 smpl(0,"Modelo M/M/1");   /* inicializa o processo de simulação */
 servidor=facility("servidor",1); /* define o servidor */
 schedule(1,0.0,cliente);  /* gera um evento chegada(1) */
 /*   Implementação do modelo */
 while (time() = = T_execucao)
 {
  cause(&evento,&cliente);  /* retira o evento da cabeca da lista */
  switch(evento)
  {
   case 1: /* chegada */
      schedule(2,0.0,cliente);
      schedule(1,expntl(T_chegada),cliente);
      break;
    case 2: /* requisita o servidor */
     if (request(servidor,cliente,0) = = 0) then
        schedule(3,expntl(T_servico,cliente);
      break;
  case 3: /* libera servidor */
     release(servidor,cliente);
     break;
   }
  }
 report();
}
```

Figura 3.5: As Fase no Processo de Execução

3.4.3 A Execução

O Gerador utiliza paralelamente as informações contidas no Gabarito e as tabelas de especificações para a geração do produto final. São utilizadas durante todo o processo de geração do código a tabela Modelo e a lista Recursos. As demais tabelas são utilizadas por poucas primitivas, sendo armazenadas em arquivos e preparadas pelo Gerador quando necessárias.

No Gerador toda o processo de geração de código é controlado por um procedimento principal (Le_Gabarito()) que identifica os comandos do gabarito (precedidos pelo caracter '%') decidindo qual primitiva deve ser executada. Identificada, a primitiva é executada, e o controle retorna ao procedimento Le_Gabarito, para continuar o processo de geração.

Cada uma das primitivas que compõem o Gerador é responsável por criar um trecho específico no código final, utilizando as informações contidas nas diversas tabelas. A figura 3.6 mostra o fluxo de dados no Gerador.

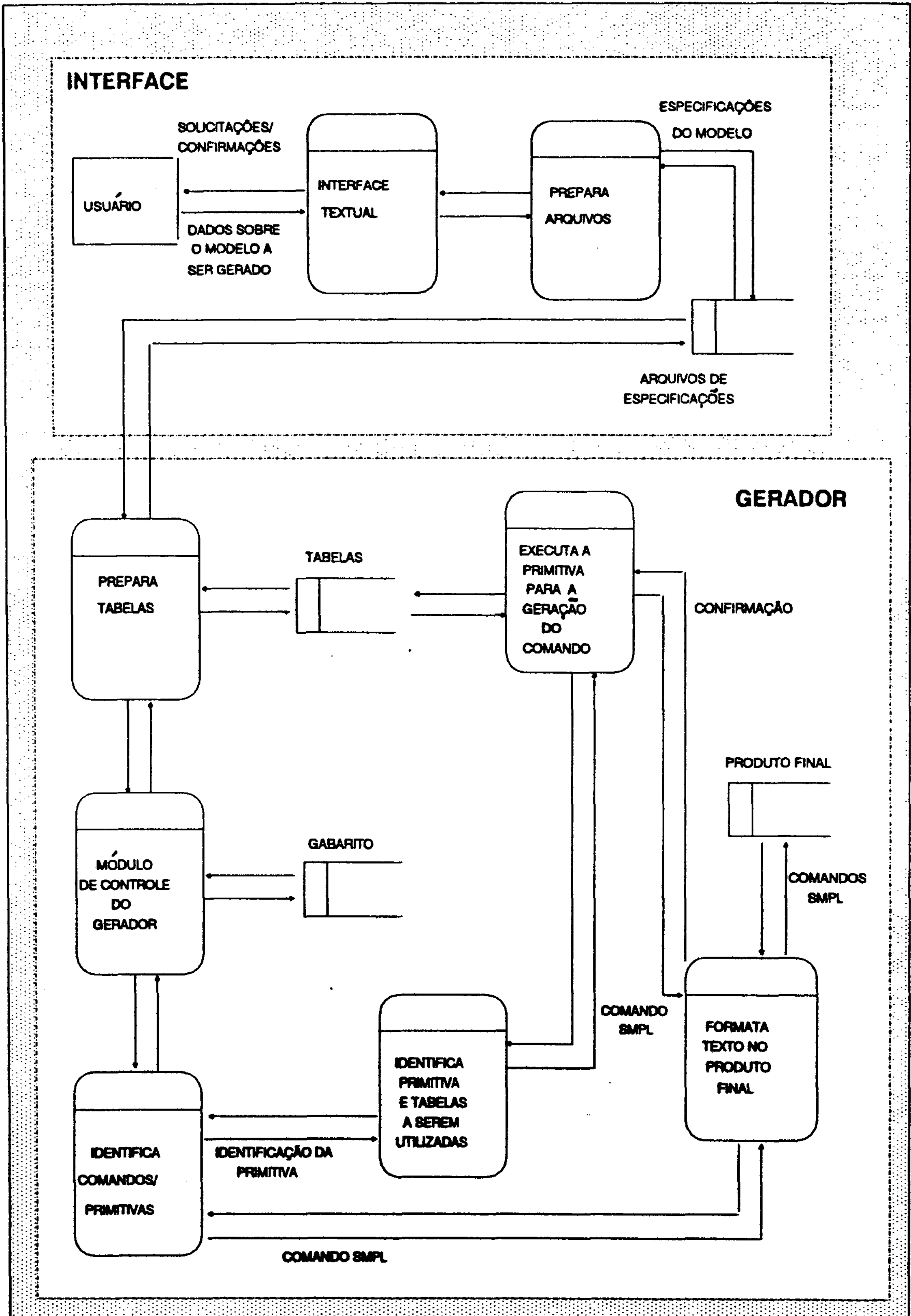

Figura 3.6: Fluxo de Dados no Gerador

São apresentados a seguir exemplos do processo de geração de código para alguns comandos do SMPL, em cada uma das três fases definidas no item 3.4.2.

Fase de Definições

Nessa fase são geradas todas as estruturas de dados necessárias à execução do programa de simulação. As informações são obtidas através da tabela Modelo e outras informações mais específicas na tabela Recursos.

1) **Gera_Var_Server()** cria todos os identificadores para os recursos do modelo. Estes identificadores serão utilizados posteriormente no programa de simulação como descritores para os diversos recursos criados.

2) **Define_Bmeans()** define as variáveis para a utilização do método de análise Batch Means.

Fase de Preparação do Modelo

Durante esta fase, são geradas todas as primitivas de preparação do ambiente e informações genéricas para a sua execução. Estas informações são obtidas através da tabela Modelo, e outras informações mais específicas na tabela Recursos.

1) *função smpl(m, s);*

*int m; char *s;*

objetivo: prepara o ambiente para a execução da simulação, limpando todas as estruturas de dados.

parâmetros: *m* = 0: não ativa a interface mtr

= 1: ativa a interface mtr

s = ponteiro para o nome do modelo

O Gerador identifica no gabarito um comando para a geração deste comando no produto final; a primitiva Nome_Parametro() é executada. Supondo que o nome do

modelo seja "Teste" e o parâmetro m seja 0 (para a não ativação da interface mtr), o resultado da geração será o seguinte:

smpl(0, "Teste");

2) *função trace(n);*

int n;

objetivo: gerar informações adicionais para a depuração dos programas.

parâmetros: n = 0 trace desativado, não será implementado

= 1 as mensagens de depuração são geradas continuamente

= 2 pausa entre as telas de mensagens geradas

= 3 pausa após a exibição de cada mensagem de depuração

= 4 atualiza contadores de linha e página

Com a identificação no gabarito de um comando para a geração dessa primitiva no produto final, a primitiva Gera_Trace() é executada. Supondo que o usuário deseje a inclusão desta ferramenta no seu programa, com mensagens de depuração geradas continuamente na tela (parâmetro n igual a 2), a saída será a seguinte:

trace(2);

3) *função f = facility(s, n);*

*char *s; int n;*

objetivo: criar e dar nome a um recurso, retornando um descritor para ser utilizado em outras operações.

parâmetros: s = ponteiro para o nome do recurso, utilizado para identificar o mesmo no relatório padrão e nas mensagens de deputação e erros.

n = número de servidores do recurso

As informações necessárias para a definição de cada um dos recursos que compõem o modelo estão na tabela Recursos. Para um modelo com dois recursos,

identificados como CPU e DISK, cada um com apenas um servidor, a saída no produto final será, após a execução da primitiva Gera_Def_Server():

Server1 = facility("CPU",1);

Server2 = facility("DISK",1);

Fase de Implementação

Nesta fase são geradas todos os eventos que compõem o programa de simulação, além das estatísticas solicitadas. As informações necessárias são obtidas através das diversas tabelas especificadas.

1) *procedimento schedule(ev, te, tkn)*

int ev,te; real te;

objetivo: escalonar o evento indicado por *ev* para ocorrência no tempo determinado pelo tempo corrente de simulação adicionado a *te*. Os eventos são escalonados em uma lista ordenada ascendentemente pelo tempo de ocorrência dos eventos.

parâmetros: *ev* = identificação do evento a ser escalondo

te = tempo para ocorrência do evento

tkn = identificação do "token" (cliente)

Os comandos para o escalonamento de eventos são gerados pela primitiva Gera_Eventos(). Supondo que o evento 1 represente a chegada do primeiro cliente ao sistema (temmpo 0.0), identificado por Customer, o seguinte comando será gerado no programa de simulação:

schedule(1, 0.0, Customer);

2) ***função report();***

objetivo: gerar um relatório com informações estatísticas sobre a simulação.

Como essa função não exige parâmetros, o usuário necessita apenas especificar se deseja ou não a emissão do relatório. Supondo que a informação fornecida pelo usuário através do editor gráfico seja positiva, o resultado da geração será o seguinte comando:

```
report();
```

4. Exemplos

São apresentados alguns exemplos de programas de simulação produzidos pelo Gerador.

4.1 Exemplo 1

Nesse exemplo a simulação é limitada por tempo e total de clientes que deixam o sistema. O usuário solicitou apenas o relatório padrão do SMPL.

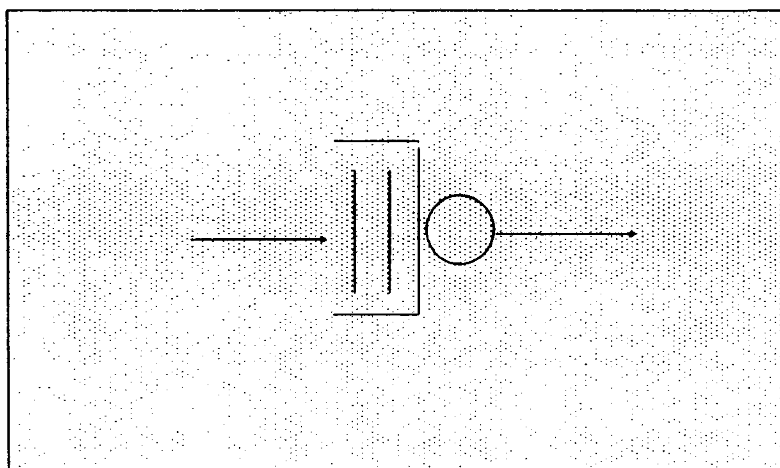

Figura 4.1: Modelo com 1 recurso

```
/* ---------------------------------------------------------------- */
#include "stdio.h"
#include "conio.h" /* para rotinas de tela e teclado     */

main()
{
/* definicoes */
float Te = 1000000.0;
unsigned int Maximo_Entidades = 0, Num_Max_Entidades = 5000;
int Event = 1, Customer = 1, Aleatorio;
real Ta1 = 200.0, Ts1 = 100.0;
int Server1;
unsigned int Total_Clientes = 0;
```

```
/* prepara o sistema de simulacao e da nome ao modelo */
 smpl(0," Exemplo");

/* cria e da nome as facilidades */
 Server1 = facility("Server1",1);

/* escalona a chegada do primeiro cliente */
 schedule(1,0.0, Customer);

while ( (time()<Te) && (Maximo_Entidades<Num_Max_Entidades) )
  {
  cause(&Event,&Customer);
  switch(Event)
   {
   case 1:
     schedule(2,0.0,Customer);
     schedule(1,expntl(Ta1),Customer);
     break;
   case 2:
     if (request(Server1,Customer,0) = = 0)
      schedule(3,expntl(Ts1),Customer);
     break;
   case 3:
     release(Server1, Customer);
     Maximo_Entidades + +;
     break;
   }
  }

 /* gera o relatorio da simulacao */
 report();
}
/* ------------------------------------------------------------------ */
```

4.2 Exemplo 2

Para este modelo foram especificas seqüências para a geração de n úmeros aleatórios, além do relatório com informações sobre o total de clientes na fila do primeiro recurso.

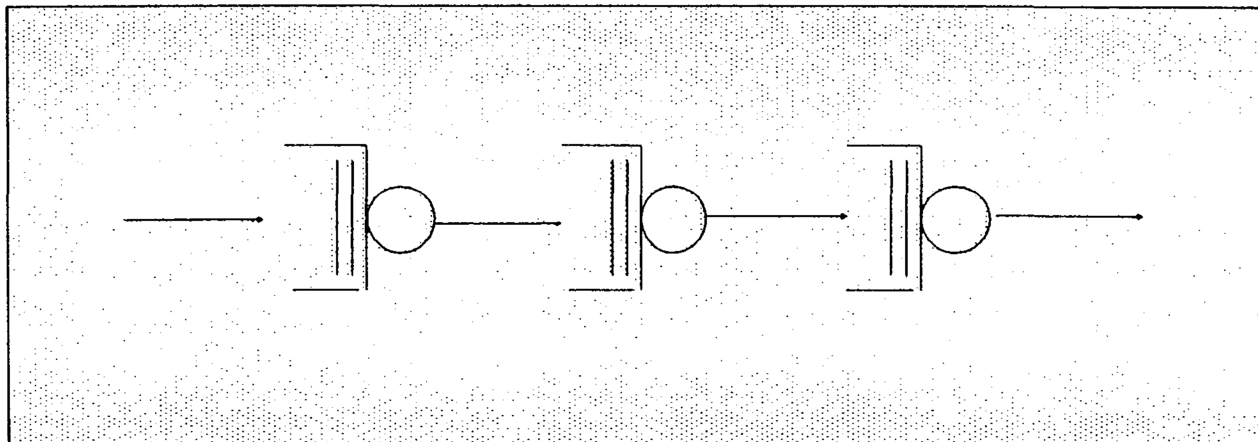

Figura 4.2: Modelo com Três Recursos

```
/* --------------------------------------------------------------------- */
#include "stdio.h"
#include "conio.h" /* para rotinas de tela e teclado      */

main()
{
 /* definicoes */
 float Te = 10000.0;
 int Event = 1, Customer = 1, Aleatorio;
 real Ta1 = 20, Ts1 = 10, Ta2 = 25, Ts2 = 15,
      Ta3 = 30, Ts3 = 20;
 int Server1, Server2, Server3;
 unsigned int Max1 = 0, Min1 = 1000;
 unsigned int Total_Clientes = 0;

 /* prepara o sistema de simulacao e da nome ao modelo */
 smpl(0," Exemplo");
 /* cria e da nome as facilidades */
```

```
Server1 = facility("Recurso1",1);
Server2 = facility("Recurso2",1);
Server3 = facility("Recurso3",1);

/* escalona a chegada do primeiro cliente */
schedule(1,0.0, Customer);

while ( (time()<Te) )
 {
 cause(&Event,&Customer);
 switch(Event)
 {
  case 1:
   schedule(2,0.0,Customer);
   schedule(1,expntl(Ta1),Customer);
   break;
  case 2:
   Total_Clientes = inq(Server1);
   if (Total_Clientes>Max1)
    Max1 = Total_Clientes;
   else
    if (Total_Clientes<Min1)
     Min1 = Total_Clientes;
   if (request(Server1,Customer,0) == 0)
    schedule(3,expntl(Ts1),Customer);
   break;
  case 3:
   release(Server1, Customer);
   schedule(4, 0.0, Customer);
   break;
  case 4:
   stream(2);
   if (request(Server2,Customer,0) == 0)
    schedule(5,expntl(Ts2),Customer);
   break;
  case 5:
   release(Server2, Customer);
   schedule(6, 0.0, Customer);
   break;
  case 6:
   if (request(Server3,Customer,0) == 0)
```

```
    schedule(7,expntl(Ts3),Customer);
   break;
  case 7:
   release(Server3, Customer);
   break;
 }
}

/* gera o relatorio da simulacao */
report();
getche();
clrscr();
printf(" \n  Relatório - Mximo e Mínino das Filas \n \n ");
printf("\n Máximo clientes Recurso 1 : %d ",Max1);
printf("\n Mínimo clientes Recurso 1 : %d ",Min1);
}
/* ------------------------------------------------------------------ */
```

4.3 Exemplo 3

Nesse exemplo, o cliente executa um ciclo no primeiro recurso antes de prosseguir para a utilização do próximo recurso. Também é utilizado o método de análise "Batch Means"

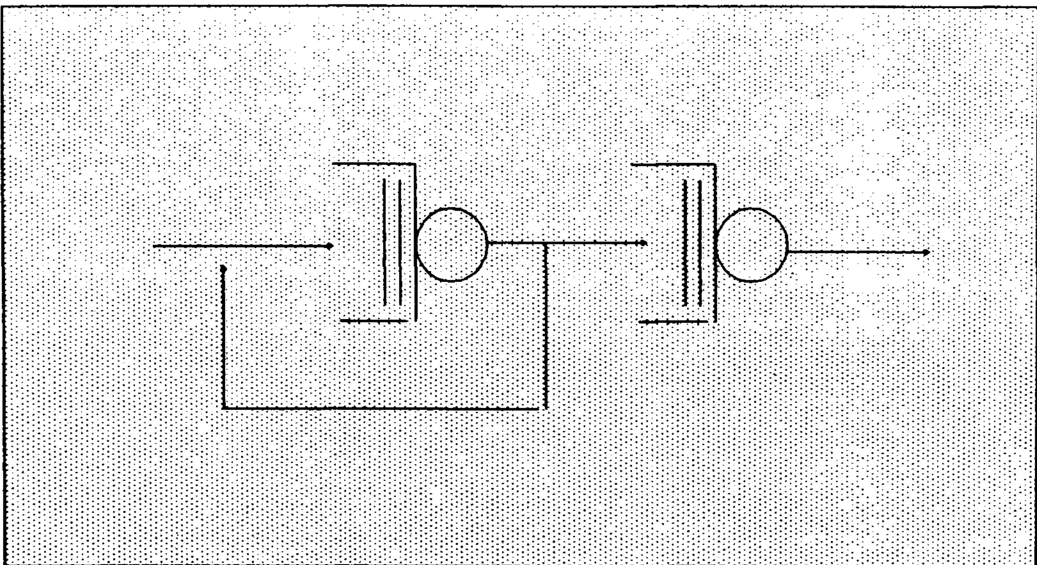

Figura 4.3: Modelo com Ciclo

```
/* ---------------------------------------------------------------------- */
#include "stdio.h"
#include  "conio.h"  /* para rotinas de tela e teclado  */

main()
{
 /* definicoes */
 float Te = 1000000.0;
 int Event = 1, Customer = 1, Aleatorio;
 unsigned int N11 = 20;
 real Ta1 = 30, Ts1 = 10, Ta2 = 40, Ts2 = 30;
 int Server1, Server2;
 unsigned int Total_Clientes = 0;
 float TBatch = 0;
 int r = 0;

 /* prepara o sistema de simulacao e da nome ao modelo */
 smpl(0," Exemplo");
 init_bm(10,100);

 /* cria e da nome as facilidades */
 Server1 = facility("Server1",2);
 Server2 = facility("Server2",1);

 /* escalona a chegada do primeiro cliente */
 schedule(1,0.0, Customer);

 while ( (time()<Te) && (r = = 0) )
  {
   cause(&Event,&Customer);
   switch(Event)
   {
    case 1:
     schedule(2,0.0,Customer);
     schedule(1,expntl(Ta1),Customer);
     break;
    case 2:
     if (request(Server1,Customer,0) = = 0)
     {
      TBatch = expntl(Ts1);
      r = obs(TBatch);
```

```
    schedule(3, TBatch, Customer);
   }
   break;
  case 3:
   release(Server1, Customer);
   if (N11)
   {
    N11--;
    schedule(2, 0.0, Customer);
   }
   else
    schedule(4, 0.0,Customer);
   break;
  case 4:
   if (request(Server2,Customer,0) = = 0)
   {
    TBatch = expntl(Ts2);
    r = obs(TBatch);
    schedule(5, TBatch, Customer);
   }
   break;
  case 5:
   release(Server2, Customer);
   break;
 }
}

/* gera o relatorio da simulacao */
report();
}
/* ---------------------------------------------------------------- */
```

4.4 Exemplo 4

Nesse exemplo a escolha do caminho a ser percorrido no sistema é feita probabilisticamente.

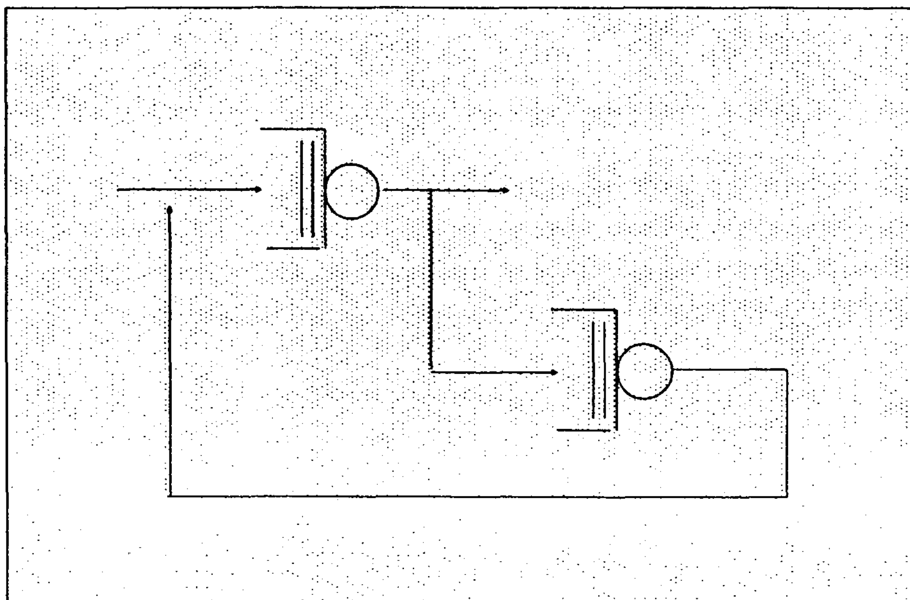

Figura 4.4: Escolha Probabilística

```
/* ----------------------------------------------------------------- */
#include "stdio.h"
#include "conio.h" /* para rotinas de tela e teclado     */

main()
{
/* definicoes */
float Te = 10000.0;
unsigned int Maximo_Entidades = 0, Num_Max_Entidades = 1000;
int Event = 1, Customer = 1, Aleatorio;
real Ta1 = 30, Ts1 = 20, Ta2 = 30, Ts2 = 40;
int Server1, Server2;

/* prepara o sistema de simulacao e da nome ao modelo */
smpl(0," Exemplo");

/* cria e da nome as facilidades */
Server1 = facility("CPU",1);
Server2 = facility("DISK",1);

/* escalona a chegada do primeiro cliente */
schedule(1,0.0, Customer);
while ( (time()<Te) && (Maximo_Entidades<Num_Max_Entidades) )
```

```
{
 cause(&Event,&Customer);
 switch(Event)
 {
  case 1:
   schedule(2,0.0,Customer);
   schedule(1,expntl(Ta1),Customer);
   break;
  case 2:
   if (request(Server1,Customer,0) = = 0)
    schedule(3,expntl(Ts1),Customer);
   break;
  case 3:
   release(Server1, Customer);
   Aleatorio = random(1,10);
   if ( (1 < = Aleatorio) && (Aleatorio > = 5) )
    Maximo_Entidades + +;
   if ( (6  < = Aleatorio) && (Aleatorio  > = 10) )
    schedule(4, 0.0, Customer);
   break;
  case 4:
   if (request(Server2,Customer,0) = = 0)
    schedule(5,expntl(Ts2),Customer);
   break;
  case 5:
   release(Server2, Customer);
   schedule(2, 0.0, Customer);
   break;
  }
 }

 /* gera o relatorio da simulacao */
 report();
}
/* ------------------------------------------------------------ */
```

4.5 Exemplo 5

No exemplo abaixo existe a posse simultânea de recursos, ou seja, o cliente utiliza o recurso identificado por PP paralelamente ao DISCOA ou DISCOB.

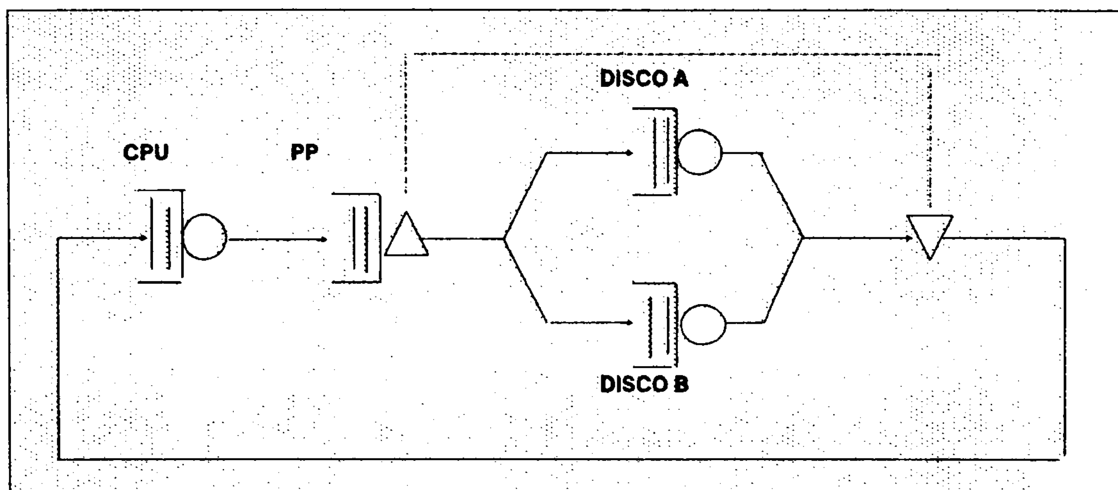

Figura 4.5: Modelo com Posse Simultânea de Recursos

```
/* ----------------------------------------------------------------------- */
#include "stdio.h"
#include "conio.h"  /* para rotinas de tela e teclado      */

main()
{
 /* definicoes */
 unsigned int Num_Voltas = 0, Num_Max_Voltas = 500;
 int Event = 1, Customer = 1, Aleatorio;
 real Ta1 = 20, Ts1 = 20, Ta2 = 20, Ts2 = 10,
      Ta3 = 30, Ts3 = 20, Ta4 = 30, Ts4 = 10;
 int Server1, Server2, Server3, Server4;
```

```
/* inicializa o sistema de simulacao e da nome ao modelo */
smpl(0," Exemplo");

/* cria e da nome as facilidades */
Server1 = facility("CPU",1);
Server2 = facility("PP",3);
Server3 = facility("DiscoA",1);
Server4 = facility("DiscoB",1);

/* escalona a chegada do primeiro cliente */
schedule(1,0.0, Customer);
schedule(1,10.0, Customer);
schedule(1,15.0, Customer);

while (Num_Voltas  < Num_Max_Voltas)
 {
  cause(&Event,&Customer);
  switch(Event)
  {
   case 1:
    schedule(2,0.0,Customer);
    break;
   case 2:
    if (request(Server1,Customer,0) = = 0)
     schedule(3,expntl(Ts1),Customer);
    break;
   case 3:
    release(Server1, Customer);
    schedule(4, 0.0, Customer);
    Num_Voltas+ +;
    break;
   case 4:
    if (request(Server2,Customer,0) = = 0)
     schedule(5,0.0,Customer);
    break;
   case 5:
    Aleatorio = random(1,2);
    if (Aleatorio = = 1)
    if (status(Server3) = = 0 ) /* servidor livre */
    {
    if (request(Server3,Customer,0) = = 0)
```

```
   schedule(6,expntl(Ts3),Customer);
   }
   else
    schedule(7,0.0,Customer);
   if (Aleatorio = = 2)
   if (status(Server4) = = 0 ) /* servidor livre */
   {
   if (request(Server4,Customer,0) = = 0)
    schedule(8,expntl(Ts4),Customer);
   }
   else
    schedule(9,0.0,Customer);
  break;
 case 6:
   release(Server2, Customer);
   release(Server3, Customer);
   schedule(2, 0.0, Customer);
   break;
 case 7:
   release(Server2, Customer);
   schedule(4,expntl(Ta2),Customer);
   break;
 case 8:
   release(Server2, Customer);
   release(Server4, Customer);
   schedule(2, 0.0, Customer);
   break;
 case 9:
   release(Server2, Customer);
   schedule(4,expntl(Ta2),Customer);
   break;
 }
}

/* gera o relatorio da simulacao */
report();
}
/* ------------------------------------------------------------ */
```

5. Considerações Finais

A utilização de geradores de aplicação para automatizar o processo de geração de código apresenta inúmeras vantagens (relatadas no item 2.2.2). Com o desenvolvimento do Gerador, foi possível confirmar que as principais contribuições da utilização de geradores ocorrem na fase de desenvolvimento de "software", com o aumento da produtividade e qualidade do código produzido e redução de custos, e durante a manutenção, o uso do Gerador possibilita que as alterações necessárias sejam feitas apenas na especificação.

Porém existem alguns pontos negativos citados na bibliografia [LEW90, LUK86, MEI91a]:

. há uma tendência dos programas gerados apresentarem processamento mais lento, devido há alta estruturação;

. na prática os geradores criam apenas parte de um sistema, pois nem sempre a automação completa do processo de produção de "software" é a solução mais eficiente e econômica [LEW90];

. existe uma limitação relativa à representação de modelos grandes e complexos, ou seja, normalmente as aplicações não podem ser descritas de modo eficiente.

Com o desenvolvimento do Gerador, foi possível observar que a estruturação dos programas gerados não prejudica sua eficiência, uma vez que os computadores se tornam cada vez mais rápidos.

Observou-se também que, apesar das restrições impostas pelos modelos simbólicos, como o modelo de rede de filas, é possível aperfeiçoar esses modelos, incorporando novos símbolos e restrições que permitam a representação de características complexas existentes nos sistemas reais. Com a utilização desses modelos e uma estruturação eficiente das informações obtidas para alimentar o gerador, é possível gerar o código que implemente fielmente o sistema especificado.

6. Referências Bibliográficas

[CHA78] CHANDY, K. M.; SAUER, C. H. Approximate Methods for Analysing Queueing Network Models of Computing Systems. **Computing Surveys**, v. 10, n. 3, September 1978.

[JAC82] JACOBSON, P. A.; LAZOWSKA, E. D. Analyzing Queueing Networks with Simultaneous Resource Possession. **Communications of the ACM**. v. 25, n. 2, p. 142-151, February 1982.

[LEW90] LEWIS, T. Code Generators. **IEEE Software**, v. 7, n. 3, p. 67-70, May 1990.

[LUK86] LUKER, P. A.; BURNS, A. Program Generator and Generation Software. **The Computer Journal**. v. 29, n. 4, p. 315- 321, 1986.

[MAC75] MACDOUGALL, M. H. System Level Simulation - **em Digital System Design Automation: Languages, Simulation and Data Base**. Computer Science Press, Inc., p. 1-115, 1975.

[MAC87] MACDOUGALL, M. H. **Simulating Computer Systems, Techniques and Tools**. The MIT Press, 1987.

[MAR80] MARYANSKY, F. J. **Digital Computer Simulation**. Hayden Book Company, Inc., 1980.

[MAS93] MASIERO, P. C.; MEIRA, C. A. A. **The Journal of Systems and Software**, v. 23, n. 1, October 1993.

[MEI91a] MEIRA, C. A. A. **Sobre Geradores de Aplicação**. São Carlos, 1991. Dissertação (mestrado) - Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo.

[PRE87] PRESSMAN, R. S. **Software Engineering - a practitioner's approach**. New York, McGraw-Hill, 1987.

[SCH86] SCHWETMAN, H. CSIM: a C-based, process-oriented simulation language. In: 1986 Winter Simulation Conference, 1986. **Proceedings**. p. 387-396, 1986.

[SHA88] SHARMA, R.; ROSE, L. L. Modular Design for Simulation. **Software-Pratice and Experience**, v. 18, n. 10, p. 945-966, October 1988.

[SOA90] SOARES, L. F. G. **Modelagem e Simulação Discreta de Sistemas.** VII Escola de Computação, São Paulo, 1990.

[SPO93a] SPOLON, Roberta. **Um Gerador de Aplicação para um Ambiente de Simulação Automático.** São Carlos, 1993. Mini-Dissertação (mestrado) - Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo.

[SPO93b] SPOLON, Renata. **Um Editor Gráfico para um Ambiente de Simulação Automático.** São Carlos, 1993. Mini-Dissertação (mestrado) - Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo.

[TAN94] TANIR, O.; SEVINC, S. Defining Requirements for a Standard Simulation Environment. **IEEE Computer**, v. 27, n. 2, February 1994.

7. Bibliografia

BORLAND C+ + Tools and Utilities Guide. Borland International, Inc., 1991.

BORLAND C + + User's Guide. Borland International, Inc., 1991.

BORLAND C+ + Programmer's Guide. Borland International, Inc., 1991.

KERNIGHAM, B. W. e RITCHIE, D. M. **C a linguagem de Programação.** 3 ed., Rio de Janeiro, Campus, 1987.

MEIRA, C. A. A.; MASIERO, P. C. Um Gerador de Aplicação para Sistemas Reativos. In: V SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, Ouro Preto-MG, 1991. **Anais.** p. 45-59.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Data Structures using** C. Englewood Cliffs, Prentice-Hall, 1990.

WALRAND, J. **An Introduction to Queueing Networks.** Prentice-Hall International, Inc., 1988.

NOTAS DO ICMSC

Serie computaçao

Nº 010/94 SAWAKI, J.; MONARD, M.C.; RODRIGUES, S.R. SABNAG: um sistema baseado em conhecimento para suporte aos u suarios da biblioteca NAG

Nº 009/94 NICOLETTI, M.C.; MONARD, M.C. Learning restricted Horn clauses: some considerations on the ij-determination concept

Nº 008/94 TOME, M.F.; DUFFY, B. GENSMAC: a numerical method for solving unsteady non-newtonian free surface flows

Nº 007/94 TOME, M.F.; MCKEE, S. Numerical simulation of viscous fluid: buckling of planar jets

Nº 006/94 MASIERO, P.C.; OLIVEIRA, M.C.F.; GERMANO, F.S.R; PIERRI, G. Authoring and searching in dynamically growing hypertext data bases

Nº 005/94 NICOLETTI, M.C.; MONARD, M.C. Limiting the background knowledge in inductive logic programming

Nº 004/93 NICOLETTI, M.C.; MONARD, M.C. Learning horn clauses using the ILP system GOLEM

Nº 003/93 ARENALES, M.N.; MORABITO, R.N. An and/or graph approach to the solution of two-dimensional non-guillotine cutting problems

Nº 002/93 NICOLETTI, M.C.; MONARD, M.C. Herbrand interpretation model and least model within the framework of logic programming

Nº 001/93 MORABITO, R.; ARENALES, M.N. An and/or graph approach to the container loading problem