DESENVOLVIMENTO DE UM AMBIENTE VISUAL E INTERATIVO PARA MODELAGEM E PROCESSAMENTO DE SIMULAÇÃO A EVENTOS DISCRETOS USANDO A ABORDAGEM DCA – TRÊS FASES.

Ricardo Miyashita

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE PRODUÇÃO.

Aprovada por:

_____

Prof. Mário Jorge Ferreira de Oliveira, Ph.D.

_____

Prof. Eduardo Saliby, Ph.D.

_____

Prof. Virgílio José Martins Ferreira Filho, D.Sc.

_____

Prof. Silvio Hamacher, D.Sc.

_____

Prof. Luiz Ricardo Pinto, D.Sc.

RIO DE JANEIRO, RJ - BRASIL.

JUNHO DE 2002

MIYASHITA, RICARDO.

Desenvolvimento de um Ambiente Visual e Interativo para Modelagem e Processamento de Simulação a Eventos Discretos Usando a Abordagem DCA – Três Fases [Rio de Janeiro] 2002

XI, 131 p. 29,7 cm (COPPE/UFRJ, D. Sc., Engenharia de Produção, 2002)

Tese – Universidade Federal do Rio de Janeiro, COPPE.

1. Simulação
2. Diagrama de Ciclo de Atividades
3. Modelagem

I. COPPE/UFRJ II. Título (série)

Dedicatória

À minha mãe.

Ao meu pai *(in memoriam)*.

Agradecimentos

Gostaria de agradecer a todos os que ajudaram neste trabalho. Em primeiro lugar aos que serviram de apoio através de teses anteriores, em concreto os autores do Simul e do Simin, precursores do trabalho aqui apresentado.

De um ponto de vista técnico foi de fundamental importância a colaboração do Prof. Silvio Hamacher que apresentou uma metodologia que norteou grande parte deste trabalho.

Agradeço àqueles que nos ajudaram tirando dúvidas e dando sugestões, em especial os colegas Eugênio, Guilherme Calôba, Maurício, Caio e Alexandre Diniz.

Uma menção especial a todo o pessoal da Coppe e do Coppead pelas incontáveis ajudas prestadas ao longo destes anos de amizade; e ao pessoal da UERJ pelo apoio nestes poucos meses de convivência.

Por fim um imenso agradecimento aos amigos que, mesmo não compreendendo os aspectos técnicos do trabalho apoiaram com muito alento e uma certa dose de paciência as longas horas passadas em frente ao computador até que este trabalho chegasse ao fim.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

DESENVOLVIMENTO DE UM AMBIENTE VISUAL E INTERATIVO PARA MODELAGEM E PROCESSAMENTO DE SIMULAÇÃO A EVENTOS DISCRETOS USANDO A ABORDAGEM DCA – TRÊS FASES

Ricardo Miyashita

Junho/2002

Orientadores: Eduardo Saliby
Mário Jorge Ferreira de Oliveira

Programa: Engenharia de Produção

Este trabalho descreve a elaboração de um novo ambiente de modelagem visual e interativo para a análise e solução de problemas de simulação a eventos discretos. O objetivo principal é o de fornecer um ambiente que possibilite a criação de modelos de simulação utilizando como base três elementos: o Diagrama de Ciclo de Atividades como filosofia de modelagem, o Método das Três Fases como algoritmo de simulação e a programação orientada a componentes. Através deste ambiente o usuário pode criar o diagrama lógico do problema a ser estudado diretamente na tela do computador. Uma vez montado o modelo é possível rodar a simulação imediatamente a partir do modelo lógico sem a necessidade de se processar nenhum código de programação escrita. Há um depurador que avisa ao usuário eventuais erros de modelagem e de preenchimento das propriedades. Os resultados da simulação são mostrados na forma de relatórios estatísticos e histogramas. Há a possibilidade de adaptar o código do programa para problemas que necessitem de ajustes especiais para a sua modelagem. O uso do ambiente mostra que ele possui características inovadoras e que fornece resultados precisos.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

DEVELOPMENT OF A VISUAL INTERACTIVE ENVIRONMENT FOR MODELLING AND PROCESSING OF DISCRETE EVENT SIMULATION USING THE ACD-THREE PHASE APPROACH

Ricardo Miyashita

June/2002

Advisors: Eduardo Saliby
Mário Jorge Ferreira de Oliveira

Department: Production Engineering

This work describes the development of a new visual interactive computational environment created to solve discrete-event simulation problems. The main objective is to provide an environment to the users that creates simulation models using three basic elements: the Activity Cycle Diagram (ACD) as the modelling phylosophy, the Three-Phase Method as the simulation algoritm and a component-oriented programming code. Using this environment, the user can build a logic diagram of the problem directely in the computer screen. Once the diagram is finished, it is possible to run the simulation immediatly using only the logic model without the need of any other written programming code. There is also a syntax checker that prompts the user of any kind of errors found in the model construction or in each block's properties. The simulation results are shown by statistical reports and histograms. There is also the possibility to customize the program code to solve some special and unusual problems. The application of this environment shows that it has innovative characteristics and precise results.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1
Introdução

A Pesquisa Operacional (PO) tem avançado tem avançado muito nos últimos anos no que diz respeito aos métodos estocásticos de modelagem, em especial da Simulação. Isto se deve ao problema prático de que muitos dos problemas derivados de uma situação real não possuirem soluções analíticas adequadas. Outro fator importante é a melhoria do desempenho dos computadores atuais que por um lado diminui o tempo de processamento dos programas Um terceiro fator de importância está ligado à engenharia de software que criou avançadas ferramentas, dentre as quais se destaca a programação orientada a componentes que permite o desenvolvimento de programas com alta complexidade de um modo integrado e robusto.

O ramo da PO tratado especificamente neste trabalho é a chamada Simulação a Eventos Discretos. Para a solução de um problema deste tipo de simulação, costuma-se dividi-lo em duas etapas mais básicas, uma de modelagem e outra de processamento. A literatura cita várias técnicas que já foram criadas para a modelagem, dentre as quais citar como as mais importantes o Diagrama de Processos e o Diagrama de Ciclo de Atividades (DCA). Sobre o agoritmo para o processamento da simulação, as técnicas mais utilizadas são a Abordagem por Processos e o Método das Três Fases. Existem muitos programas de simulação que utilizam interfaces gráficas de modelagem para a criação de Diagrama de Processos e que realizam o processamento através do algoritmo baseado na abordagem por Processos (Martinez, 2001).

Uma outra abordagem é a da escola inglesa de simulação, que costuma utilizar o Diagrama de Ciclo de Atividades (DCA) em conjunto com o Método das Três Fases. Podemos denominar esta abordagem simplificadamente de DCA-Três Fases. Verifica-se que a escola inglesa, embora seja seguida por muitos pesquisadores, dispõe de uma gama de programas muito menor do que a escola americana. E se analisarmos os poucos programas existentes, observamos ainda que carecem de uma interface de modelagem amigável, tornando a tarefa de criação do modelo algo demorado, trabalhoso e sujeito a erros. Já os modernos programas da escola americana de simulação possuem estas interfaces integradas ao simulador em suas versões mais modernas, eliminando desta forma os problemas citados. Vemos por trás desta lacuna uma oportunidade para o desenvolvimento de uma nova ferramenta, que consideramos

ser de grande utilidade para os profissionais da área de simulação. O presente trabalho busca ir ao encontro desta necessidade. Detalhamos os objetivos e as etapas do estudo a seguir.

O objetivo deste trabalho é a elaboração de um novo ambiente para a modelagem e processamento de simulação a eventos discretos com o desenvolvimento interativo do **Diagrama de Ciclo de Atividades** (DCA) do problema estudado diretamente na tela do computador e o processamento automático do algoritmo de simulação segundo o **Método das Três Fases** sem a necessidade da criação de um código de programação intermediário. A abordagem de programação será **orientada a componentes**, segundo as modernas tendências da engenharia de software e que são seguidas pelos mais recentes sistemas de simulação a eventos discretos.

Pretende-se atender com este sistema tanto os profissionais de simulação quanto os acadêmicos, principalmente aqueles acostumados com o uso do Diagrama de Ciclos de Atividades.

Os capítulos da tese foram divididos segundo as etapas de desenvolvimento adotadas nesta pesquisa.

No capítulo 2 são apresentados os conceitos teóricos envolvidos sobre o assunto, mostrando a importância do tema dentro do estado da arte em simulação. São descritas também as opções que foram adotadas para o sistema, no que se refere a três aspectos importantes em qualquer sistema de simulação: (1) a metodologia de modelagem lógica, (2) o algoritmo de processamento e (3) a programação orientada a componentes. O restante do segundo capítulo descreve em detalhe a metodologia de modelagem lógica baseada no Diagrama de Ciclo de atividades.

O capítulo 3 trata de uma das bases conceituais do sistema desenvolvido, que é a programação orientada a componentes.

O algoritmo de processamento da simulação denominado de Método das Três Fases é descrito no capítulo 4.

O capítulo 5 descreve o ambiente de modelagem. É descrito o funcionamento do sistema completo, juntamente com alguns exemplos básicos. Mostram-se os principais recursos do sistema, destacando-se os relatórios e histogramas.

Dedicamos todo o capítulo 6 para mostrar exemplos de aplicação do ambiente desenvolvido, de modo a facilitar uma melhor visualização dos recursos disponíveis.

O capítulo 7 descreve os passos que o programa segue para importar as informações do modelo lógico e passá-las para a estrutura de dados necessária para o

processamento da simulação. São descritos em detalhe os passos desta análise, seguindo a mesma seqüência de operações que são realizadas pelo computador.

Por fim são apresentadas as conclusões obtidas no último capítulo, bem como as sugestões para trabalhos futuros dentro da mesma linha de pesquisa.

Nota: ao longo do texto indicaremos entre parênteses e em itálico alguns termos técnicos em inglês mais utilizados pelos especialistas de computação, ao lado dos respectivos termos em português, com o fim de facilitar a compreensão do texto, visto que os tradutores brasileiros nem sempre possuem um termo único para um correspondente termo em inglês, e o termo em português utilizado pelo autor poderia levar a confusões ou equívocos.

Capítulo 2
A importância do tema de estudo dentro do campo da simulação a eventos discretos

2.1 A simulação a eventos discretos

A simulação é um ramo da Pesquisa Operacional. Segundo a definição de Banks (1999) consiste na "imitação da operação de um processo ou de um sistema da vida real ao longo do tempo. Quer seja feita à mão ou através de um computador, a simulação envolve a geração de uma história artificial com o objetivo de obter inferências relativas às características operacionais do sistema real".

Há diversos tipos de simulação. Neste trabalho enfocaremos a chamada simulação a eventos discretos, que "é a modelagem de sistemas em que as variáveis de estado somente mudam de valor em pontos discretos ao longo do tempo." (idem)

Como este texto trata somente de simulação a eventos discretos, todas as vezes que nos referirmos ao termo simulação, não o faremos em relação ao termo geral, mas ao tipo específico correspondente aos eventos discretos.

O mundo da simulação está em constante transformação. Ele tem avançado cada vez mais na busca de novas soluções tanto no campo acadêmico quanto no empresarial. A advinda de computadores de processamento cada vez mais rápido agilizou o processamento das rodadas de simulação, e os programas ganharam popularidade.

Antes da criação das interfaces gráficas para computadores, os problemas de simulação eram modelados através da codificação de comandos escritos em um arquivo de texto. Os programas de simulação criados para uso em sistemas operacionais que dispõem de uma interface gráfica (por exemplo, Windows) permitem criar o modelo de simulação de um modo mais rápido, eliminando em grande parte a tarefa de codificação manual dos programas tradicionais. Denominamos estes programas como sendo de modelagem visual, pois a criação do modelo é realizada através de ícones dos elementos do modelo que são "montados" na tela do computador. À medida que os blocos representativos das entidades, filas e atividades são acrescentados na tela, há uma "geração" automática do correspondente objeto dentro da estrutura interna do

algoritmo de simulação, operação esta que facilita muito a tarefa de modelagem para o usuário.

Uma recente pesquisa envolvendo os membros da Sociedade Britânica de Pesquisa Operacional (Hlupic, 2000) mostram que a grande maioria dos usuários, tanto do meio universitário quanto industrial já utiliza sistemas que possuem interfaces gráficas “com facilidade de uso e bons recursos visuais” para a modelagem de problemas de simulação.

Os problemas tradicionais, das áreas de utilização mais comuns de utilização, são facilmente modelados através destes programas. Por exemplo, *softwares* de simulação de modelagem visual de uso mais comum no Brasil como Arena, Automod e Promodel apresentam boas soluções para problemas nas áreas de manufatura, logística, serviços de atendimento e outros ligados à gerência de operações.

Os programas de modelagem puramente visual, no entanto, apresentam menos flexibilidade se comparados aos programas feitos com linhas de código, pois enquanto os primeiros são dedicados a alguns tipos de aplicação, os programados manualmente podem adaptar-se aos problemas de áreas de aplicações especiais e que possuem peculiaridades que fogem da modelagem padrão. Esta queixa sobre a limitação de flexibilidade dos programas de modelagem visual é comum entre os analistas de simulação (Joines e Roberts, 1998, Roberts e Dessouky, 1998, Kamigami e Nakamura, 1996). O problema da falta de flexibilidade foi atenuado pela introdução de módulos programáveis nas versões mais recentes dos programas de modelagem visual. Estes módulos podem ser acrescentados ao modelo de modo a adaptá-lo ao problema real. No entanto, verifica-se que freqüentemente estes módulos costumam tornar o processamento da simulação mais lento, pois geralmente trabalham com instruções que são interpretadas ao durante a simulação.

Outro problema é que muitos os programas comerciais de modelagem visual não permitem a visualização do seu código interno nem a alteração do algoritmo que conduz a simulação, o que seria importante para verificar a precisão dos resultados encontrados e para implementar modificações para problemas específicos. Muitas inovações surgem na pesquisa acadêmica na área de simulação, envolvendo novas técnicas de amostragem, algoritmos de processamento, formas de acompanhamento e ferramentas de análise dos dados de entrada e de saída. Nos programas comerciais estas melhorias podem ser realizadas somente pelos próprios fabricantes, e os pesquisadores

da área acadêmica ficam carentes de instrumentos adequados para implementar pessoalmente as novas técnicas de simulação tão logo surjam as inovações.

2.2 Requisitos gerais do sistema proposto

Procuramos desenvolver neste trabalho um ambiente de modelagem e processamento de simulação que procurasse conter as principais características desejáveis citadas pela literatura. Por exemplo, Hlupic (2000) sugere alguns requisitos importantes para um bom programa de simulação: "facilidade de aprendizado e uso, baixo custo, excelentes recursos gráficos, flexibilidade de extender além das funções básicas e por fim ferramentas inteligentes de modelagem do experimento e análise de resultados". Em nosso sistema conseguimos abordar praticamente todos os aspectos citados, embora não se tenha alcançado a máxima perfeição em cada um deles.

A importância do tema pode ser vista também através de outro artigo que relata um painel realizado na última versão do *Winter Simulation Conference*, o mais importante congresso científico sobre simulação. O título do trabalho é "Simulation Enviromnent for the New Millenium" e recolhe algumas opiniões interessantes de alguns especialistas sobre quais seriam as características importantes para os sistemas de simulação (Kachitvichyanukul et al., 2001). Um dos participantes do painel, C. Dennis Pedgen propõe que "o que os usuários querem são ferramentas poderosas e flexíveis e, além disso, fáceis de aprender e de utilizar". Diz ainda que "quando olhamos para o futuro da simulação, uma das idéias mais promissoras é o conceito de se ter modelos pré-construídos ou componentes de modelo que possam ser *conectados entre si* de modo a formar um modelo de seu sistema". Todas estas características: flexibilidade, facilidade de aprendizado/utilização e uso de componentes visuais estão presentes na nossa proposta.

Buscamos neste estudo de um modo especial uma solução que procurasse aliar as vantagens dos sistemas de modelagem visual com as dos sistemas de programação manual. Buscamos obter conjuntamente **facilidade de modelagem** através de uma interface amigável, **flexibilidade para alteração do código** e **eficiência computacional** no processamento da simulação. Pretendemos também criar programas segundo uma **arquitetura aberta,** podendo ser conseqüentemente modificáveis nos níveis de modelagem e de programação. Como forma de implementação e de

estruturação utilizamos a **programação orientada a componentes**, seguindo as tendências atuais dos programas de simulação.

O desenvolvimento de um sistema que possuísse as características acima citadas poderia ser feito através de várias formas de implementação. Citamos a seguir as possíveis alternativas para a modelagem lógica e para o algoritmo de execução da simulação que poderiam ser adotadas. Ao final são mostradas as alternativas escolhidas para o ambiente de simulação aqui desenvolvido.

2.3 Modelagem lógica dos problemas de simulação

2.3.1 Modelagem por Diagrama de Ciclos de Atividades

A análise de um problema através de simulação envolve uma modelagem lógica do problema estudado. A modelagem de um problema constitui um dos três pontos estratégicos para a acurácia de uma simulação, juntamente com os dados e o experimento (Robinson, 1999).

Dentre as diversas metodologias de modelagem que poderiam ser escolhidas, escolhemos para o nosso ambiente um tipo que é também bastante utilizado por diversos outros profissionais de simulação, inclusive no Brasil e que se chama **Diagrama de Ciclos de Atividades** ou **DCA.** Ele foi idealizado inicialmente por Tocher (1963) e é seguido pela chamada escola "inglesa" de simulação. A descrição de como é montado um diagrama DCA pode ser encontrada nos textos de autores que o utilizam (Pidd, 1998; Pinto, 1999; Odhabi et al., 1998 e Tavares *et al.*, 1996). Sobre os elementos que fazem parte do diagrama há pequenas variações que dependem do autor, mas o núcleo desta metodologia está bem definido na literatura.

Podemos utilizar os DCA para descrever um problema através dos estados das entidades em cada momento. Em sua versão mais simples, os elementos do DCA são três: **entidades, filas e atividades**. Estes elementos devem ser colocados no diagrama dentro de regras básicas. A primeira regra diz que deve haver alternância entre atividades e filas dentro de um ciclo. A segunda regra diz que uma determinada fila pode conter somente um tipo de entidade. Uma terceira regra que diz que as entidades devem percorrer ciclos fechados. Há outras regras de construção, mas as três acima citadas são as principais.

O resultado final de um trabalho de modelagem segundo o DCA são vários ciclos interligados, que representam o comportamento do sistema de um ponto de vista lógico. A tarefa principal do analista do problema é abstrair da situação estudada quais são as entidades, atividades e filas que melhor representam o problema, bem como as ligações que formam os diversos ciclos. Um exemplo de diagrama DCA é mostrado na figura 2.1. Trata-se do problema do atendente de teatro apresentado por Pidd (1998). Um funcionário que trabalha na bilheteria de um teatro e recebe tanto a pessoas que querem comprar um ingresso quanto a ligações de cliente solicitando informações. Duas fontes representam a origem de entidades no sistema: uma fonte de pessoas e uma fonte de telefonemas. Há no exemplo quatro atividades: Chegada de novas ligações, Atendimento destas ligações, Chegada de pessoas e Atendimento de pessoas. E são três as filas existentes: Aguarda Atendimento pessoal, Aguarda Atendimento telefônico e Atendente Ocioso.

Note que há três ciclos neste sistema, um para a entidade Telefonema (à esquerda, com setas mais grossas), um para o Atendente (no centro) e um para a entidade Pessoa (à direita, com setas tracejadas). Estes três ciclos se encontram nas atividades em que é necessária a cooperação de duas entidades para sua execução; é o caso da atividade Atende Pessoa, que para sua execução necessita que o atendente esteja ocioso e que haja alguma pessoa na fila de atendimento. Fica claro neste exemplo que há uma alternância entre filas e atividades dentro de cada ciclo.

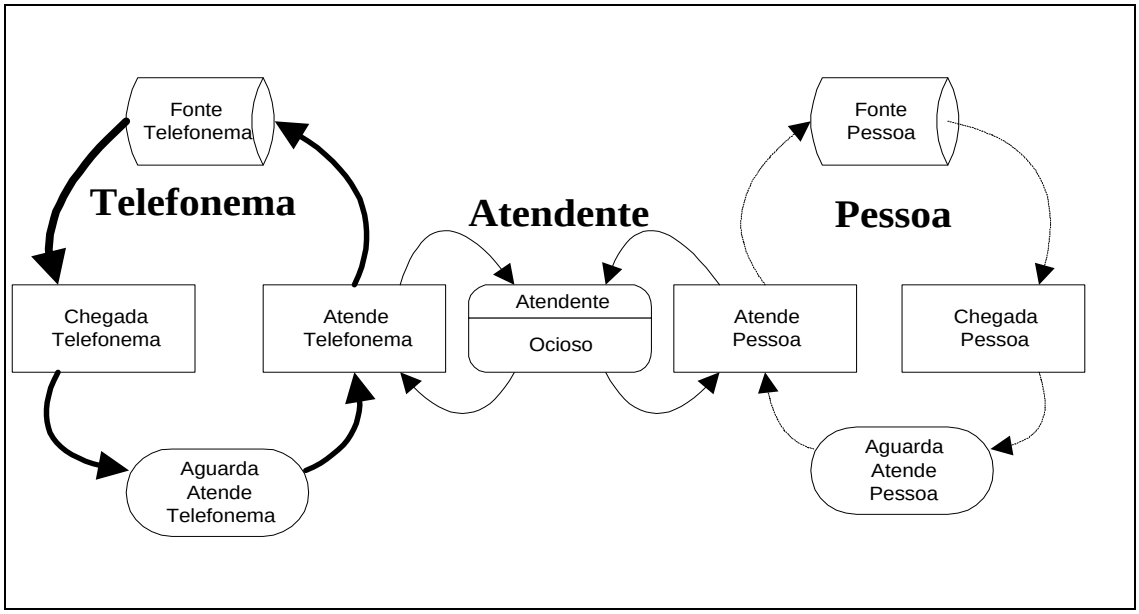


Figura 2.1 Diagrama lógico do tipo DCA para o problema do atendente de teatro, baseado em Pidd (1998).

Os DCAs se destacam por sua facilidade de compreensão, pois são construídos a partir de elementos simples, e também pela flexibilidade em representar problemas de diversas naturezas, podendo modelar várias situações como, por exemplo, filas de atendimento, sistemas de produção, sondas de perfuração de petróleo até navios em um porto. Os diagramas DCA facilitam a criação de programas de simulação, pois a partir deles é fácil visualizar os Eventos, que são peças importantes para descrever o comportamento do modelo. No caso do exemplo do teatro os eventos são seis, um para cada atividade de chegada e dois para cada uma das outras atividades (um correspondente ao seu início e outro ao seu fim).

Ao longo do seu desenvolvimento, que envolveu uma série de pesquisadores, o DCA demonstrou muitas qualidades e algumas limitações. Com o fim de contornar estas limitações, foram desenvolvidas algumas variações sobre o DCA (ou ACD, como é chamado em inglês), que são o DCA expandido (Pinto, 1999), o X-ACD (Pooley e Hugues, 1991), o SH-ACD (Odhabi et al., 1998) e o H-ACD (Kiembaum e Paul, 1994). O próprio exemplo que foi apresentado, o do atendente de teatro, possui um elemento que não consta nas primeiras versões do DCA, que é elemento fonte. A fonte é importante para representar a origem de novas entidades do sistema, e está presente em todas as implementações mais modernas do DCA que foram citadas acima.

De uma forma geral, o DCA tem se mostrado extremamente útil, pelas seguintes razões:

1) Possibilita a criação de modelos com uma lógica bem definida;

2) Mostra os ciclos das entidades de modo bastante claro;

3) Reflete de uma maneira completa as informações necessárias para a construção de programas para a execução simulação no computador.

A principal alternativa ao DCA para a modelagem lógica é a modelagem por processos, que será descrita logo adiante. O DCA constitui uma metodologia melhor que a por processos para problemas em que há um grande número de entidades interagindo nas atividades e também quando há regras complexas para definição para início de atividades. Estas razões fazem com que muitos pesquisadores prefiram utilizar o DCA (Martinez e Ioannou, 1999; Pidd, 1998; Shi, 1997). Um fator negativo para o DCA é o de que o usuário deve raciocinar dentro de uma lógica que lhe é bastante peculiar, pensando em termos de filas e atividades e às vezes tendo que criar artifícios especiais para adaptar estes elementos para modelar o problema estudado.

2.3.2 Modelagem por processos

Um segundo tipo de modelagem lógica é por **processos**, muito conhecido por influência dos pesquisadores americanos que a utilizam. É uma abordagem mais intuitiva para problemas de fluxo de materiais, como por exemplo, na linha de montagem de um determinado produto, em que as peças são processadas em seqüência através de várias máquinas. O diagrama de um processo utiliza os símbolos próprios para representar este fluxo, desde a criação da entidade no sistema, passando pelos processos que percorre até a saída do sistema. Os três módulos principais utilizados são: **Criação, Processo e Saída**. Há muitos mais módulos que estão disponíveis nos programas que utilizam este tipo de modelagem. A figura 2.2 mostra o exemplo de um modelo lógico utilizando a abordagem por processos, criado para o simulador Arena.

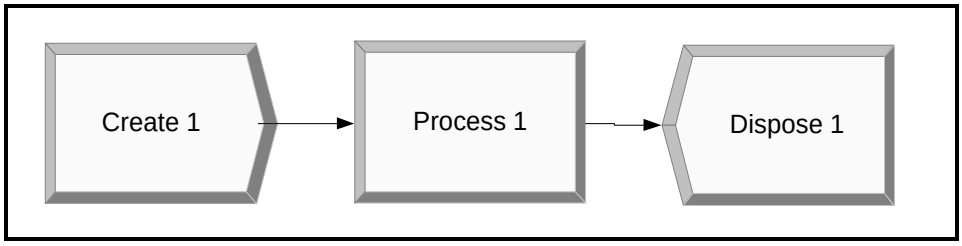


Figura 2.2 Modelo lógico de um problema de simulação utilizando a representação por processos.

A modelagem por processos se diferencia da modelagem por ciclos de atividades porque nos processos as entidades não percorrem necessariamente ciclos que se fecham.

Em ambientes de modelagem por processos, outros módulos podem ser acrescentados aos três principais, como por exemplo, um que represente um desvio condicional após um determinado processo, em que, dependendo do estado da entidade, ela pode ser direcionada para um caminho ou outro dentro do modelo. Estes módulos vão variar de acordo com o programa de simulação utilizado. Estes módulos ou blocos geralmente são agrupados por famílias nos simuladores, dependendo do fim para os quais foram desenvolvidos. Abaixo mostramos dois exemplos de grupos de blocos do Arena, um contendo elementos básicos ou *Basic Process* (fig 2.3) e outro contendo módulos de transporte ou *Advanced Transfer* (fig 2.4).

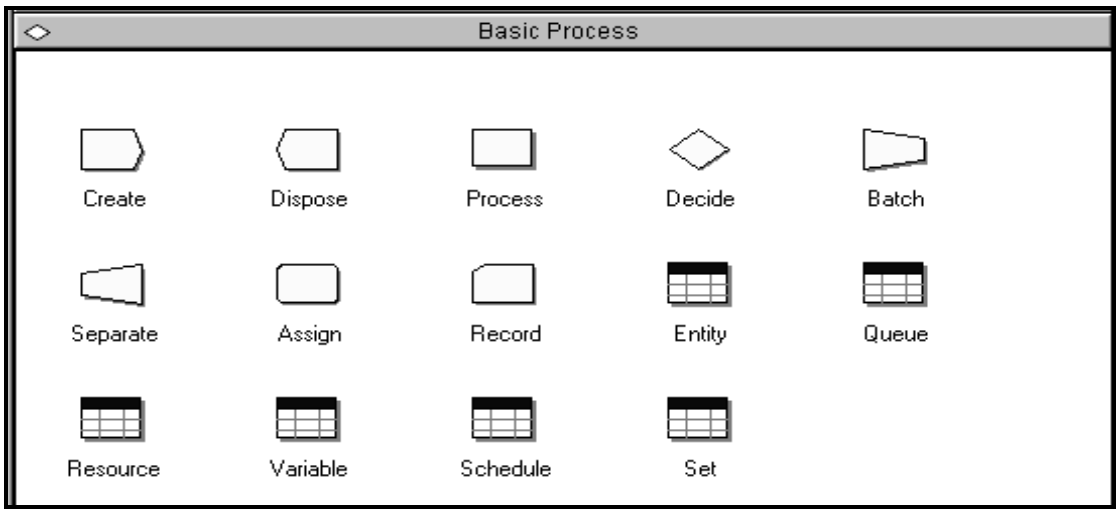


Figura 2.3 Painel básico contendo blocos de modelagem por processos utilizados pelo Arena.

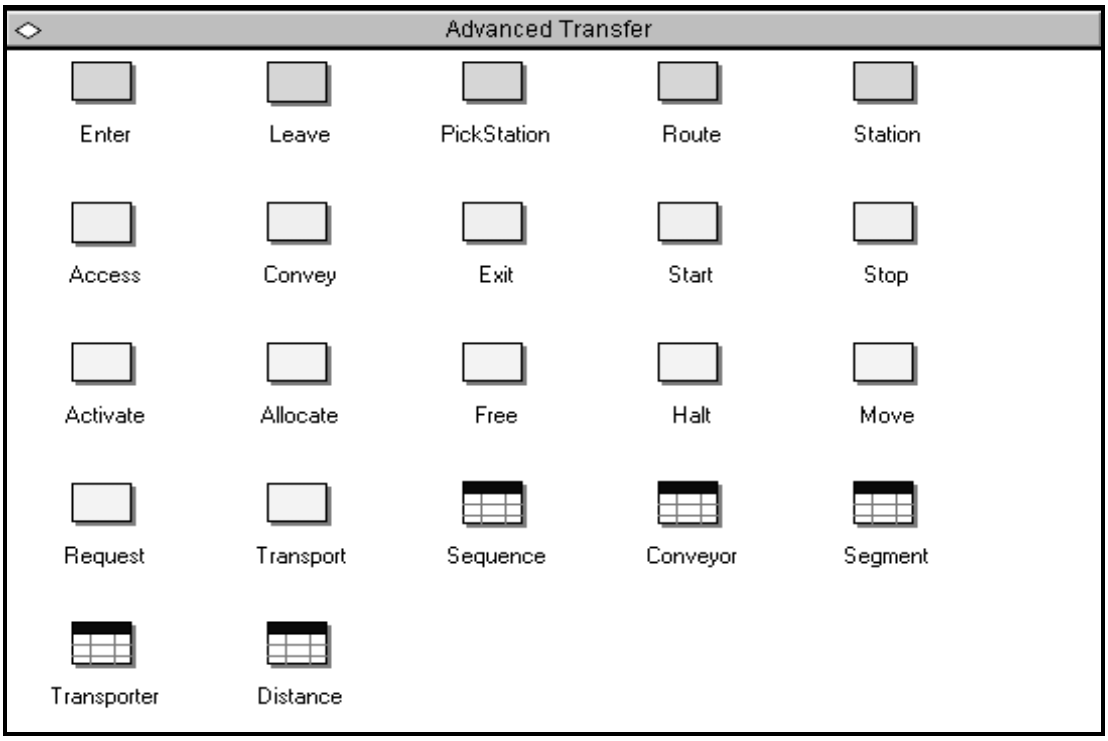


Figura 2.4 Painel avançado contendo blocos de transporte para modelagem por processos utilizados pelo Arena.

2.3.3 Considerações sobre as metodologias de modelagem lógica.

Na criação de um sistema completo de simulação a eventos discretos é necessário optar por uma dentre as opções existentes para a modelagem da lógica, sendo as duas mais conhecidas o DCA e a modelagem por processos. Estas duas abordagens são quase que igualmente capazes de modelar a maioria dos problemas comuns de

simulação, apesar de que, dependendo do tipo de problema, haverá vantagens em se utilizar determinado tipo de representação ao invés do outro.

Há algumas outras metodologias de modelagem existentes, mas que não foram analisadas com mais detalhe por não representarem alternativas úteis do ponto de vista prático para os problemas que pretendemos estudar e por serem de uso mais restrito. São exemplos deste tipo de metodologia as redes de Petri e os Diagramas de Estado.

No caso do sistema que foi desenvolvido neste trabalho, descrito nos capítulos posteriores, optamos por utilizar o Diagrama de Ciclo de Atividades por se estar mais familiarizado com o seu uso e pelas três características citadas anteriormente (cfr 2.3.1). Acredita-se, no entanto, que não haveria nenhum problema em se utilizar a representação por processos caso esta tivesse sido escolhida.

Cabe ainda notar que tem havido uma certa convergência na definição dos módulos representativos dos elementos de modelagem lógica tanto para os adeptos do Diagrama de Ciclo de Atividades quanto para os que utilizam modelagem por processos. Ambos têm utilizado um conjunto bastante semelhante de blocos para modelagem. Vemos dois exemplos claros a seguir. O primeiro foi criado por Kiembaum (1994 e 1995) com a idéia de melhorar o DCA, onde foram acrescentados alguns blocos para construção de modelos mais complexos; embora tenha sua origem no DCA, o conjunto de elementos é na verdade representativo de modelagem por processos. O segundo é o conjunto de blocos utilizados pelo Arena para modelagem básica. As versões mais antigas do Arena (p.ex. na versão 3) utilizavam blocos mais ligados ao algoritmo de processamento por interação de processos, como por exemplo: Enter, Server e Leave. Estes blocos estão agora agrupados dentro do bloco Process. Veja na tabela 2.1 a semelhança dos dois conjuntos, o H-ACD de Kiembaum e o Basic Process do Arena: praticamente são os mesmos elementos.

Tabela 2.1 Símbolos comuns entre duas metodologias de modelagem lógica, ambas utilizando a abordagem por processos.

H-ACD (Kiembaum, 1995)	Basic Process (Arena 5, 2001)	Função
Source	Create	Fonte
Sink	Dispose	Término
Delay	Process	Atividade
Branch	Decide	Desvio Condicional
Batch	Batch	Juntar em lote
UnBatch	Separate	Separar lote
Assign	Assign	Definir atributo
Queue	Queue (embutido em Process)	Fila

Nosso trabalho está inserido dentro deste mesmo contexto de convergência, apresentando como base o Diagrama de Ciclo de Atividades, mas aproveitando blocos que tradicionalmente são utilizados na modelagem por processos, como os dois apresentados anteriormente.

2.4 Algoritmo de processamento da simulação

Além da possibilidade de escolha entre diferentes modos de representação lógica, é necessário definir o tipo de processamento ou algoritmo de simulação a ser utilizado na construção do programa. Ao longo do tempo foram criados vários destes algoritmos para a simulação a Eventos Discretos. O desenvolvimento e uso dos programas de simulação fizeram com que, na prática, houvesse uma concentração em duas abordagens básicas: a abordagem por Interação de Processos e a Abordagem por Atividades pelo Método das Três Fases.

2.4.1 Abordagem por interação de processos

Esta é provavelmente a abordagem mais utilizada atualmente pelos fabricantes de programas de simulação de uso mais disseminado (cfr Pidd, 1998). A abordagem por **Interação de Processos** *(Process Interaction)* analisa o comportamento das entidades como fluxo, desde o nascimento até a morte da entidade dentro do modelo, e é

composto por pequenas operações em seqüência. As entidades percorrem o fluxo determinado por estes processos, tendo dois tipos de “retardos” em seu caminho: retardos condicionais e retardos incondicionais. O processamento dos eventos relacionado com as operações é administrado através de uma lista de eventos futuros ou FEL (*future events list*) e de uma lista de eventos correntes. Algumas atividades requerem o uso de um ou mais recursos de capacidade limitada. Um mesmo recurso pode ser disputado por processos de entidades diferentes, e o gerenciamento de qual dos processos irá de fato utilizar o recurso fica a cargo do programa de simulação. Um típico programa que utiliza a abordagem por processos tem uma série de comandos relacionados com o uso de recursos como, por exemplo, na linguagem Siman (utilizada pelo Arena), onde temos SEIZE (carregamento de um recurso), DELAY (utiliza o recurso por um tempo) e RELEASE (libera o recurso). Outros programas de simulação por processos como, por exemplo, o GPSS e o SIMSCRIPT II.5 utilizam comandos com a mesma função, embora com nomes às vezes um pouco diferentes. O trabalho do programador está relacionado, portanto, com a descrição detalhada das etapas que uma entidade percorre desde a sua criação até a sua saída do sistema, passando pelos servidores em que vai entrar e declarando os recursos que vai utilizar nas diversas atividades. Esta descrição pormenorizada das operações a serem executadas é feita de uma maneira automática pelos simuladores mais modernos, bastando para o usuário criar o diagrama lógico do modelo na tela de interface gráfica. Para exemplificar o modo como são construídos os códigos de programas de simulação de abordagem por processos, mostramos quais os comandos utilizados para representar um caso simples de um servidor com uma fila:

Tabela 2.2 Comandos típicos de um simulador que utiliza a abordagem por processos.

CREATE	Cria a entidade.
QUEUE	Define a fila no qual a entidade vai permanecer até que o servidor esteja desocupado.
SEIZE	Inicia a atividade, solicitando o recurso.
DELAY	Faz a entidade aguardar até o fim do tempo da atividade.
RELEASE	Libera o recurso.
DISPOSE	Saída da entidade do servidor.

O algoritmo baseado em interação de processos avança as entidades pelas atividades do sistema enquanto não encontrar motivos para parar este avanço. Chamamos de *Delay* ou "retardo" ao tempo que a entidade deve aguardar um servidor disponível sem avançar no seu caminho. Internamente, para o processamento da simulação, o programa separa dois tipos de "retardos", os de tipo Condicional e os de tipo Incondicional. Os "retardos" incondicionais ocorrem quando o tempo de espera não depende de elementos externos à atividade que está sendo executada, bastando somente que o relógio da simulação avance o tempo necessário para a reativação da entidade. Já os "retardos" condicionais dependem de que determinadas condições do sistema sejam satisfeitas para que a entidade possa continuar seu caminho.

Como podemos ver, o algoritmo baseado em interação de processos tem como foco as entidades e recursos, sendo descrito em detalhe os caminhos que a entidade deve percorrer e os recursos que ela necessita para realizar as tarefas necessárias.

2.4.2 Abordagem por Varredura de Atividades e o Método das Três Fases

Na abordagem por Varredura de Atividades (*Activity Scanning*) o foco de análise está nas atividades integrantes do modelo. Cada início e fim de atividade são denominados de eventos, e as operações realizadas em cada evento são detalhadas na forma de pacotes de código simples, e que podem ser interligados segundo uma lógica clara e direta.

Na abordagem por varredura de atividades, o algoritmo de simulação avança o relógio e verifica, para cada etapa de atividade, se já chegou o momento para executá-la, e caso seja positivo, realiza as ações indicadas.

As primeiras implementações de programas que utilizaram esta abordagem apresentavam certa lentidão de processamento, pois deveriam varrer todos os eventos do modelo para verificar se estava no momento de executá-los. A fim de agilizar o processamento, foi desenvolvida uma variante da abordagem por varredura de atividades denominada **Método das Três Fases.**

Neste método, os eventos associados às atividades se diferenciam em dois tipos: os Eventos Condicionais (ou do tipo C) e os Eventos Não Condicionais (ou do tipo B). O algoritmo passa a ter três fases, chamadas de A, B e C.

Na fase A ocorre o avanço do relógio para o próximo evento a ser realizado, assim como no algoritmo original. Este avanço é feito consultando-se a tabela de tempos dos eventos futuros e pegando o que estiver mais próximo.

A segunda fase envolve os eventos do tipo B e se denomina conseqüentemente de Fase B. Nela são executados todos dos eventos B que estão programados para serem realizados neste mesmo tempo do relógio de simulação.

A terceira etapa envolve os eventos Condicionais e denomina-se Fase C. Aqui são testadas todas as condições de início das atividades e caso estas sejam verificadas, se programa o início da atividade. A condição típica de início de atividade é a presença de entidades nas filas que antecedem a atividade. Há uma economia de tempo nesta etapa, pois somente os eventos condicionais são testados, mas não os incondicionais.

Geralmente os inícios das atividades correspondem a eventos condicionais (tipo C) e os finais, a eventos não condicionais (tipo B).

Um típico programa formulado com esta abordagem possui partes definidas de código para cada um dos eventos do modelo. Este código pode ser implementado na forma de procedimentos (um para cada evento) ou na forma de objetos (como por exemplo, criando-se uma classe do tipo “EventoB”).

2.4.3 Considerações sobre os algoritmos de simulação.

Atualmente a maior parte dos simuladores utiliza como base o algoritmo por interação de processos. É o caso do Arena, ProModel, AutoMod, Extend, GPSS, SLAM e Simscript II.5 entre outros.

Os programas que utilizam o algoritmo por atividades são de uso mais concentrado em torno a grupo de pesquisas das universidades e podem se citar dentre os mais utilizados o ELSE/VS7 (derivado do VS6), o Hocus, citado em Pidd, (1998), o

Cyclone e o Stroboscope, citados por Martinez e Ioannou (1999). Os dois primeiros foram desenvolvidos por pesquisadores na Inglaterra, que é onde surgiu o DCA e os outros nos EUA, o que prova que esta metodologia tem sido bem aceita em todo o mundo.

Existem muitas outras abordagens, que são descritas na literatura especializada (Pidd, 1998; Banks, 1999), mas que são mais antigas e pouco utilizadas. Dentre estas abordagens destaca-se a abordagem por Eventos utilizada no SIMSCRIPT nas versões anteriores a II.5 (cfr Pidd, 1998).

2.5 Relação entre metodologia de modelagem e algoritmo de simulação.

No desenvolvimento de um novo sistema de simulação deve-se prever qual a metodologia de modelagem e qual o algoritmo de simulação que será utilizado. Existe uma forte tendência, como se pode verificar em quase todos os sistemas existentes no mercado, de se utilizar as seguintes composições: Modelagem por Diagramas de Ciclos de Atividades em conjunto com o processamento através de Varredura de Atividades; e Modelagem por Processos em conjunto com o algoritmo de Interação de Processos. Estas são composições que seguem uma tendência natural, pois em ambos os casos se utilizam termos semelhantes tanto para a modelagem quanto para o algoritmo: no primeiro caso “atividade” e no segundo caso “processo”. De fato, estas duplas estão unidas por um motivo histórico, pois os seus desenvolvimentos foram realizados em paralelo. Em outras palavras, a abordagem por Varredura de Atividades “casa” bem com o DCA enquanto que o algoritmo por interação de processos “casa” bem com a modelagem por processos.

É preciso notar que esta dependência não é absoluta, e que a princípio nada impede que haja sistemas com outros tipos de configuração. É o caso do trabalho de Shi (1997) que trabalha com um simulador que utiliza o algoritmo de interação de processos (no caso o SLAM) e com a metodologia de modelagem por Diagramas de Ciclos de Atividades. Embora o DCA utilizado seja um pouco diferente daquele que é comum entre os outros pesquisadores (mostra somente as atividades, mas não as filas do sistema), o trabalho parece ser bem sucedido.

No nosso trabalho optamos por utilizar o algoritmo de Varredura de Atividades em sua versão do Método das Três Fases em conjunto com uma variante do DCA

tradicional contendo elementos adicionais. O algoritmo foi escolhido porque possibilita a criação de módulos de programação de fácil manipulação. Já a metodologia de modelagem possui a vantagem de possibilitar a criação de modelos complexos. O DCA utilizado possui elementos adaptados da modelagem por processos, como é o caso de desvios condicionais e de atividades que utilizam recursos. Nosso trabalho se enquadra, portanto dentro desta perspectiva de convergência entre modelagem por processos e por Ciclos de Atividades.

2.6 As três bases do sistema proposto

O sistema proposto neste trabalho possui três níveis fundamentais: como filosofia de modelagem utiliza o **Diagrama de Ciclo de Atividades**, tem como algoritmo de processamento o **Método das Três Fases** e é implementado através de **Componentes de simulação**. Estes elementos estão dispostos no sistema em partes bem definidas, e seu inter-relacionamento pode ser analisado graficamente através da figura abaixo. Note que formam um sistema com três camadas interligadas.

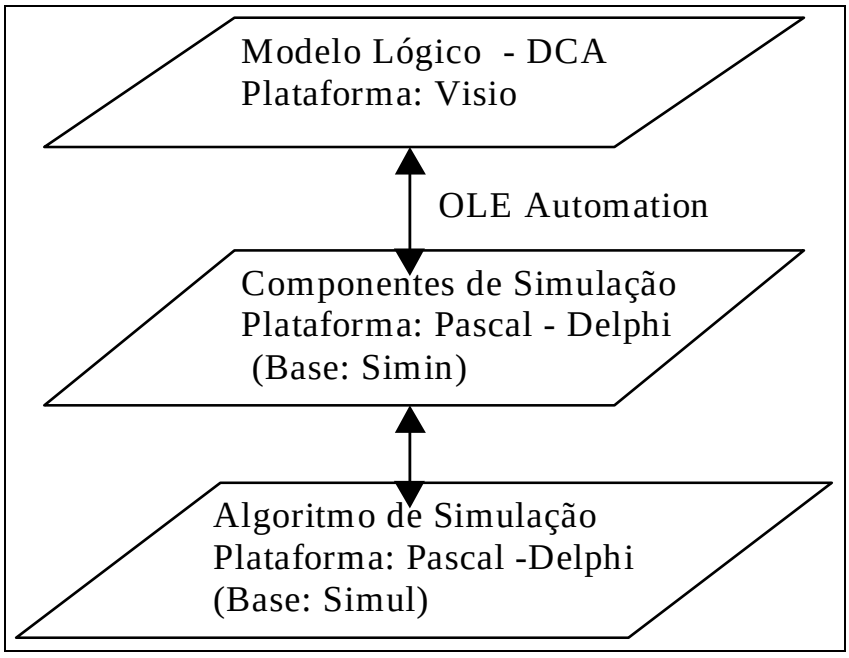


Figura 2.5 Relacionamento entre as três bases principais do sistema: Modelagem lógica pelo DCA (no Visio); Componentes de simulação e Algoritmo de simulação em Pascal-Delphi

O sistema desenvolvido tem o nome de **SimVisio.** Sua implementação completa envolveu dois softwares. Em primeiro lugar o compilador Delphi, onde foram desenvolvidos os componentes de simulação (fila, atividade, entidade, fonte…). Estes componentes estão baseados no que foi desenvolvido no sistema Simin (Pinto, 1997), sobre a qual foram feitas adaptações, tendo sido criados mais alguns componentes novos. Para o "motor" de simulação, ou seja, o "cérebro" do sistema e onde está implementado o algoritmo de simulação, foram aproveitadas várias rotinas de simulação desenvolvidas para o Simul (Saliby, 1995), pois o Delphi possui como linguagem base o Pascal, linguagem original do Simul. De um ponto de vista técnico, as rotinas de simulação estão localizadas em *Units* em Pascal, contendo os procedimentos do algoritmo do Método das Três Fases.

Tanto para os componentes Simin quanto para as rotinas do Simul foram necessárias uma série de adaptações e melhorias para sua utilização no SimVisio.

Outro programa utilizado para o nosso desenvolvimento foi o Visio, que é um ambiente gráfico para criação diagramas que é ao mesmo tempo rápido e eficiente. No nosso caso, foi especialmente útil por ter a capacidade de criar novos recursos a partir de outros pré-existentes. Adaptamos o Visio para o uso em simulação através da criação de uma série de blocos ou *Shapes* para cada elemento do DCA tradicional como, por exemplo, Fila, Atividade, Fonte e Conector. Foram também criados shapes que não pertencem ao DCA tradicional: FilaRec, Inspeção, Condição e Auto. Outra característica do Visio extremamente útil foi a capacidade de disponibilizar as informações internas dos Shapes para serem lidas a partir de programas externos. No nosso trabalho, as informações dos diagramas DCA criados no Visio são lidos pelo programa principal, criado em Delphi, e com base nas informações recolhidas é montada a estrutura de dados necessária para se poder rodar a simulação. Esta "conversação" entre o Visio e o programa em Delphi é feita através da tecnologia denominada de Automação OLE (*Ole Automation*), que é uma tecnologia de interface entre softwares. Detalhes sobre a leitura das informações dos diagramas são descritos na documentação do Visio.

O nosso novo sistema utiliza também os chamados **componentes**, que são estruturas de código organizadas em pequenos módulos de uma forma padronizada. O uso dos componentes em simulação permite a criação de programas robustos, com uma lógica bem definida e que são escalonáveis, pois novos elementos podem ser acrescentados com facilidade. Além disso, os módulos formados pelos componentes

podem ser substituídos por outros mais adequados para problemas específicos, bastando para tanto retirar o componente antigo e colocar o outro mais adequado em seu lugar. Com eles é possível a reutilização de partes de um modelo em outro, uma vez que componentes antigos podem ser utilizados em novos programas sem que seja necessário reprogramar o código.

Podemos ressaltar como mais uma vantagem do sistema criado o fato de que as ferramentas utilizadas possuem **baixo custo** se comparado com os pacotes comerciais de simulação de uso mais disseminado. Estas ferramentas, no nosso caso, são o compilador Delphi e a ferramenta gráfica Visio.

Dentro desta nova concepção, os modelos são criados dentro de um ambiente visual, interativo, de fácil aprendizado e utilização.

Para utilizá-lo é simples. Primeiramente são colocados na tela os blocos representativos de cada elemento do modelo; a seguir são indicadas as ligações entre os blocos, formando os ciclos de cada entidade do modelo. Por fim é acrescentado o bloco de controle da simulação. Todo o processo é realizado com uma série de facilidades: basta clicar sobre o bloco lógico para indicar suas propriedades, há cores diferentes para distinguir os diversos ciclos de entidades, há um verificador de erros de preenchimento de propriedades e um verificador de erros de modelagem.

Uma vez que foi criado todo o modelo lógico, o Diagrama de Ciclo de Atividades, basta clicar sobre o bloco de controle de simulação para executar a Simulação. Ao final do processo é possível verificar o comportamento do sistema através de relatórios estatísticos e histogramas. Os dados de cada histograma podem ser exportados para um arquivo texto para que se possa fazer análises mais aprofundadas através de pacotes estatísticos.

Há ainda a opção de programação manual (em linhas de código) para problemas com peculiaridades especiais e que fujam à modelagem tradicional.

A seguir mostramos um quadro comparativo entre o SimVisio e os outros sistemas nos quais ele é baseado, de modo a tornar visível as contribuições apresentadas pelo presente trabalho.

Tabela 2.3 Comparativo de funcionalidades entre o SimVisio e os seus precursores, o Simul e o Simin.

Item	Simul	Simin	SimVisio Puro	SimVisio + Delphi	Observações
No. de blocos lógicos	3	8	12 (*)	13 (*)	(*) 8 visuais + 4 embutidos (ent, hist, dist, atr)
Programação Manual	Sim	Sim	Não	Sim	
Programação Visual	Não	Parcial (*)	Sim	Sim	(*) não mostra ligações
Verificador de Erros de Modelagem	Não	Parcial (*)	Sim	Sim	(*) somente para alguns erros mais básicos
Orientado Objetos/Compon.	Objetos	Componentes	Componentes	Componentes	
Algoritmo	3 Fases	3 Fases	3 Fases	3 Fases	
Visualização das ligações	Não	Não	Sim	Sim	
No. arquivos gerados p/ modelo	1 (*) (1 .pas)	4 (**) (2 .dfm e 2 .pas)	1 (***) (1 .vsd)	5 (****) (1 .vsd e 2 .pas e 2 .dfm)	(*) .pas p/ Turbo-Pascal (**) .dfm e .pas p/ Delphi (***) .vsd p/ Visio (****) Visio + Delphi
Verificador de dados/ entrada	Não	Não	Sim	Sim	
Facilidade de aprendizado	Difícil (*)	Médio (**)	Fácil	Difícil (***)	(*) linguagem Pascal (**)linguagem Delphi (***)linguagem Delphi e programação p/ Visio

Capítulo 3
Componentes de Simulação

3.1 – Programação orientada a componentes

A criação de programas de simulação geralmente envolve uma grande quantidade de código computacional. A adoção de uma correta estrutura de dados pode facilitar tanto a elaboração do programa quanto a execução do mesmo pelo computador. Adotamos para o nosso sistema o uso de **componentes**, que são módulos especiais de programação. Historicamente, a introdução dos componentes no mundo da computação facilitou muito o trabalho de programação, acelerando o processo de criação de aplicativos e permitindo uma prática reutilização de partes de um programa em outro.

Os componentes são o resultado do aprimoramento do conceito de programação orientada a objeto. Os componentes são um tipo específico de objeto, e como todo objeto está dentro de uma hierarquia de classes, segundo a qual há algumas classes que são do tipo base (hierarquicamente superiores) e outras classes que são derivadas (hierarquicamente inferiores). Além disso, possui também as características gerais de herança, encapsulamento e polimomorfismo dos objetos.

Todos os objetos possuem campos (para armazenamento de dados) e métodos (procedimentos pertencentes a cada um dos objetos). Já os componentes, além dos campos e dos métodos, possuem também propriedades e eventos (Miller et al., 1998). As propriedades são atributos de aparência e de relacionamento que estão abertos para acesso a partir de outras classes e ferramentas gráficas. O acesso ao valor das propriedades de um componente pode ser realizado externamente a partir de outros programas, que inclusive pode ser escrito em uma linguagem diferente daquela no qual o componente foi criado originalmente. Por exemplo, um componente ActiveX (padrão criado pela Microsoft) pode ter suas propriedades lidas em sistemas que utilizam linguagens diversas como Delphi, C++ ou mesmo Java. Os componentes geralmente apresentam propriedades gráficas como largura, comprimento, cor, posição, etc, que os tornam próprios para serem manuseados dentro de uma janela (ou formulário) do computador durante o processo de criação de aplicativos em ambientes gráficos de desenvolvimento de programas.

Em simulação o uso de componentes nos programas tem sido cada vez maior. Pidd et al. (1999) cita vários sistemas que foram desenvolvidos com base em componentes, dentre os quais podemos citar como sendo os mais importantes o DEVS – *Discrete Event System Specification*, criado por Zeigler (1984); HLA- *High Level Architecture*; VSE – *Visual Simulation Environment*, criado por Balci (1998); e Silk, criado por Healy e Kigore (1998). Outro sistema, mais recente, baseado em componentes é o CORSA – *Component-ORiented Simulation Architecture*, criado por Chen e Szymanki (2001), e que tem como foco de aplicação alguns problemas de telecomunicações envolvendo redes de telefonia celular.

Pidd cita também as definições apresentadas por diversos autores a respeito da noção de componentes. Consideramos a mais adequada a definição apresentada por Fukunari e Wolfe (1998) segundo a qual um componente é "um elemento de software de conteúdo próprio que pode ser controlado dinamicamente e ser unido a outros de modo a formar uma aplicação".

Do ponto de vista da linguagem de implementação, há vários tipos de componentes, dentre os quais podemos citar os componentes ActiveX (padrão da Microsoft), os "Java Beans" (Miller et al., 1998) e os componentes Delphi, que podem ser utilizados não só nos compiladores Delphi mas também nos compiladores C++Builder.

Os termos classe e objeto podem ter pequenas diferenças de significado dependendo do autor ou do programa que se está utilizando. Vamos utilizar neste trabalho as denominações destes termos segundo o contexto do ambiente Delphi, segundo o qual as classes são tipos estruturados de dados, sendo que os objetos de um programa são instâncias deste tipo de classe. Em outras palavras, fazendo-se um paralelo com os tipos de variáveis (*Integer, Real, Boolean, String etc*) e as variáveis respectivas (por exemplo, A,B,C,D etc), podemos dizer que as classes se assemelham aos tipos assim como os objetos se assemelham com as variáveis. Os componentes, quando utilizados na simulação, levam à criação de programas com uma série de vantagens sobre aqueles que são feitos segundo outras metodologias mais tradicionais.

A seguir vamos analisar em detalhe cada um dos conceitos importantes dos componentes. Esta análise será feita tendo em vista a utilização dos componentes especificamente para programas de simulação. Como os componentes são objetos, veremos em primeiro lugar as características derivadas da programação orientada a objeto, que são: encapsulamento, herança e poliformismo. Além destas, os

componentes também possuem outras características próprias que vão além dos objetos em geral, e que são importantes para o uso em programas de simulação. Estas propriedades são a reutilização, a modularidade e a automação. Eis o que cada uma destas características quer dizer:

a) Herança: é a habilidade de definir uma nova classe em termos de uma ou mais classes antecessoras (Cantù, 1998). Podemos criar novas classes durante o desenvolvimento de um determinado modelo, classes estas que são correspondentes a tipos específicos de entidades, atividades e filas, bastando para isto fazer referência a estas classes originais quando forem criadas estas novas classes. As classes descendentes herdam das antecessoras os campos e os métodos, e podem se diferenciar das ancestrais por linhas de código adicionais que podem ser introduzidas nelas, comportando-se como as classes ancestrais onde não há mudança de código e tendo comportamentos distintos e próprios onde estas mudanças foram realizadas.

b) Encapsulamento: é o "termo formal usado para descrever o modo de proteger os campos e os métodos que estão dentro do objeto" (Ege, 1992). As regras de encapsulamento determinam quais as funções de um determinado objeto estarão ocultas aos outros objetos do programa e quais serão acessíveis. Assim, por exemplo, as entidades de um modelo de simulação podem "encapsular" suas propriedades de modo a restringir o acesso pelo usuário ou não. Pode-se permitir a leitura e a alteração destas propriedades somente quando isto for conveniente. Com mecanismos de restrição é possível evitar falhas acidentais durante a montagem do modelo que poderiam levar a erros na execução do programa. Este recurso permite que a comunicação entre vários objetos seja feita de modo harmônico e seguro. Há diferentes termos utilizados para determinar quais os tipos de acesso permitidos dentro de uma classe. Linguagens como C++ e Delphi utilizam os termos *Private, Public, Protect e Publicated* cujos significados são: *private* - contendo funções e campos que não são acessíveis nem aos clientes nem às subclasses, sendo de uso privado da própria classe; *protect* - seção contendo funções e campos que são acessíveis pelos clientes desta classe; *public* - tanto as funções quanto os campos desta seção são acessíveis tanto pelos clientes quanto pelas subclasses; e *publicated* - facilita a edição das propriedades, ou seja, campos de dados que podem assumir diferentes tipos padrões (*Integer, Real, Boolean etc*) ou até

tipos definidos pelo usuário, e que estão disponíveis para acesso e alteração de conteúdo tanto em tempo de execução quanto em tempo de projeto.

c) Polimorfismo: é a capacidade de se referir a objetos de classes diferentes por meio de um nome comum a ambos. Deste modo podemos realizar uma vinculação dinâmica, onde se poder trabalhar com um objeto sem determinação prévia de tipo durante o período de codificação do sistema, e de determiná-lo mais tarde, dinamicamente, ou seja, durante o processamento do programa. Geralmente há uma série de tipos de classe diferentes que podem ser escolhidos. Isto é feito segundo a conveniência do programa no momento da escolha, ou seja, enquanto é rodado. Exemplificando, suponhamos dois tipos de objeto: Fila e FilaRec. Podemos ter um nome comum de método aplicado aos dois tipos: Fila.TiraDaFila e FilaRec.TiraDaFila. Assim, podemos ter um comando Self.TiraDaFila, onde Self é um objeto que é determinado quando o programa estiver rodando. Para o caso de Self ser do tipo Fila, o comando correspondente a ser executado é Fila.TiraDaFila; se Self é do tipo FilaRec, o comando executado é FilaRec.TiraDaFila.

d) Reutilização: capacidade pela qual os componentes podem ser utilizados novamente em outros programas diferentes daquele para o qual foi criado inicialmente e mesmo ser distribuído para uso de outros programadores. A reutilização é uma característica básica dos componentes e é decorrente da forma pela qual foram projetados.

e) Modularidade: propriedade dos objetos de poderem se condensar em módulos individualizados que podem ser agrupados de acordo com as necessidades do usuário. A modularidade facilita a montagem e a leitura do código, principalmente na depuração e ampliação do programa. Na prática, cada objeto possui um código que o define, e este pacote de código é o que constitui propriamente o que chamamos aqui de módulo.

f) Automação: tecnologia que permite a utilização conjunta de um mesmo objeto por dois programas diferentes. Através da automação é possível criar e processar a simulação de um modo integrado podendo realizar a modelagem gráfica em um determinado programa e o processamento em outro. O programa gráfico disponibiliza as propriedades dos objetos e dos diagramas através de uma chamada interface de automação, e o segundo programa lê estas informações pela interface, processa as

informações e roda o algoritmo de simulação. No caso do trabalho aqui desenvolvido utilizamos o programa Visio para a criação do modelo lógico visual e o SimVisio para ler as informações e processar a simulação. O SimVisio é um programa que foi feito em Delphi com recursos de leitura de dados segundo a automação Ole. Tecnicamente se utiliza o termo "servidor" para caracterizar o papel do Visio e "cliente" para o SimVisio.

3.2 – Descrição dos componentes utilizados

Neste trabalho foram utilizados componentes para o processo de criação de programas de simulação. Para o nosso caso, utilizamos modelos criados segundo a filosofia dos Diagramas de Ciclos de Atividade (DCA), e os componentes a serem utilizados são os correspondentes aos elementos do DCA: Fila, Atividade, Fonte e outros como os componentes Distribuição e Histograma.

De um modo geral, neste trabalho foram utilizados como base alguns componentes do Simin: Atributo, Entidade, Histograma, Fila, Fila com Prioridade, Fonte, Atividade e Distribuição (Pinto, 1999).

Sobre estes componentes foram feitas algumas adaptações para melhorar a sua performance. Além disso, muitos outros componentes novos foram criados: FilaRec, Condição, Inspeção e Auto.

A seguir fazemos uma descrição de cada um destes componentes:

a) Componente TDistrib

Este componente representa as distribuições de probabilidades que serão usadas no modelo. Cada distribuição deve ser representada por um componente deste tipo. Suas propriedades são:

- *Tipo*: representa o tipo de distribuição e pode ser: Normal (μ,σ); Exponencial Negativa (β); Weibull (β,α); Poisson (λ); Erlang (β,k); Triangular (a,m,b); Uniforme (a,b); Inteira Uniforme (a,b); Bernoulli (p); Log Normal (μ,σ); Gama (β,α); Beta (α_1,α_2); Empírica Discreta (c1,x1,c2,x2, ...); Empírica Continua (c1,x1,c2,x2, ...).

- *Param_1, Param_2, Param_3*: números reais que representam os parâmetros da distribuição escolhida na ordem que foram definidos acima (definição clássica). Assim, se a distribuição for, por exemplo, Normal, o valor de *Param_1* = média (μ) e *Param_2* = desvio padrão (σ). Neste caso, o Param_3 pode ser desprezado. Da mesma forma, na Exponencial, podem ser desprezados os parâmetros 2 e 3.
- *Semente*: número inteiro que representa a semente do gerador aleatório e pode variar de 1 a 20.
- *Valores*: número inteiro que representa o número de pares de valores das distribuições empíricas. Pode ser no máximo igual a 20. Esta propriedade pode ser desprezada no caso de escolha de distribuições não empíricas.
- *X_01, Y_01 a X_20, Y_20*: números reais que representam os pares de valores das distribuições empíricas. Também podem ser desprezadas no caso de escolha de distribuições não empíricas.

b) Componente THistograma

Este componente representa os histogramas que serão usados no modelo. Cada histograma deve ser representado por um componente deste tipo. Suas propriedades são:

- *Tipo*: representa o tipo de histograma e pode ser Simples ou Integrado. O primeiro é usado para acumular duração de atividades e o último para acumular tamanho de filas.
- *Base*: número real que representa a base do histograma.
- *Largura*: número real que representa a largura das faixas do histograma.

c) Componente TEntidade

Este componente corresponde às classes de entidades físicas do modelo e apresenta somente as propriedades padrões (*Name* e *Tag*). Cada classe de entidade deve ser representada por um componente deste tipo.

d) Componente TAtributo

Este componente corresponde aos atributos das classes de entidades. Cada atributo deve ser representado por um componente deste tipo. Suas propriedades são:

- *Entidade*: classe de entidade a qual o atributo pertence.
- *Identificador*: string que identifica o atributo. Não pode ser aproveitada a propriedade *Name* do atributo, entre outros fatores, por ela ter que ser um nome único e muitas vezes, existe a necessidade de definir atributos com o mesmo nome para entidades diferentes.

e) Componente TFonte

Corresponde às fontes fornecedoras de entidades temporárias do sistema. Cada fonte deve ser representada por um componente deste tipo. Suas propriedades são:

- *Ent*: classe de entidade a qual pertencem as entidades geradas pela fonte.
- *Prioridade*: número inteiro de 1 a 9 que representa o valor de prioridade das entidades geradas. A maior prioridade tem valor 0 e a menor 9. Esta prioridade pode ser usada para retirada preferencial de determinadas entidades de filas, como será visto a seguir.

f) Componente TFila

Componente que representa as filas do sistema. Controla o acesso das entidades às atividades, organizando-as em fila. Libera estas entidades, uma a uma, quando requisitadas, conforme disciplinas de acesso preestabelecidas. Cada fila deve ser representada por um componente deste tipo. Suas propriedades são:

- *Ent*: classe de entidade a qual pertencem as entidades da fila.

- *Prioridade*: número inteiro de 1 a 9 que representa o valor de prioridade das entidades colocadas naquela fila ao início da simulação. A maior prioridade tem valor 0 e a menor 9. Como já dito, esta prioridade pode ser usada para retirada preferencial de determinadas entidades de filas, como será visto a seguir.
- *Primeiro e Ultimo*: No início da simulação as entidades permanentes são colocadas em diversas filas do sistema. Para se controlar quantas entidades serão colocadas em cada fila, são indicados os números das entidades que estarão naquela fila no início da simulação. Assim, *Primeiro* representa o número da primeira entidade a ser colocada na fila e *Ultimo* o número da última. Por exemplo, se *Primeiro*=6 e *Ultimo*=8, serão colocadas na fila, no início da simulação as entidades de números 6, 7 e 8. Se for uma só entidade, *Primeiro=Último*.
- *HistEspera*: histograma representativo do tempo de espera das entidades na fila. É um histograma do tipo Simples.
- *HistTamanho*: histograma representativo do tamanho da fila. É um histograma do tipo Integrado.

g) Componente TAtividade

Este componente representa as atividades do sistema. Cada atividade deve ser representada por um componente deste tipo. Suas propriedades são:

Distribuicao: é a distribuição de probabilidades que rege a duração da atividade.

- *Prioridade*: número inteiro que define a ordem de execução da atividade. Quanto menor o número maior a prioridade. Sendo assim, uma atividade com prioridade igual a 1 terá sua condição de início testada antes de uma com prioridade igual a 2. Pode haver duas ou mais atividades com o mesmo valor de prioridade. Neste caso, a ordem de teste de início será a ordem de criação dos componentes no formulário. A priorização só tem sentido entre atividades que disputam a mesma entidade.
- *Filas_Pre_01 a 10 e Filas_Pos_01 a 10*: Filas anteriores e posteriores à atividade. Podem existir até 10 filas de cada tipo.
- *Fonte_01 a 10:* Fontes de entidades que precedem a atividade. Podem existir até 10 fontes/sumidouros.

- *Hist*: histograma representativo do tempo de duração da atividade.

Outros componentes criados para o novo programa:

h) Componente TFilaRec

Componente que representa a origem de entidades permanentes no sistema, sendo ao mesmo tempo uma fila depois de iniciada a simulação. Suas propriedades são as mesmas que TFila:

- *Ent, Prioridade, Primeiro, Ultimo, HistEspera, HistTamanho*: mesmas características que TFila.

i) Componente TCondição

Componente que estabelece desvios condicionais para as entidades que saem de um bloco Atividade com base em um teste lógico sobre o valor de um atributo de uma entidade. Suas propriedades são:

- *NomeAtributo*: Nome do atributo cujo valor vai ser avaliado. Note que o nome do atributo deve estar previamente definido na respectiva Fonte ou FilaRec, no momento da criação da entidade a que pertence.
- *Condição*: Pode ser “>”, “<”, “<=”, “>=”, “<>” ou “=”.
- *Valor*: Valor de comparação. Por exemplo: Em “Sede >=0” o valor de comparação é 0 (zero).

j) Componente TInspecao

Componente que serve para direcionar a entidade ao final da execução de uma atividade com base em uma porcentagem de aprovação. Suas propriedades são:

- *Aprov1 a Aprov10*: número de 0 a 100 que representa a porcentagem de entidades aprovadas. Para cada faixa de aprovação há um caminho especificado dentro do diagrama.

k) Componente TAuto

Componente que realiza a busca automática da fila de menor tamanho dentre aquelas que seguem a uma atividade. Suas propriedades são:

- *Fila1 a Fila 10*: indica as filas de destino ao qual a atividade está ligada

Capítulo 4
O algoritmo de simulação

Mostraremos neste capítulo o algoritmo de processamento da simulação utilizado. Trata-se da abordagem por Varredura de Atividades (*Activity Scanning)* em uma versão que se chama o **Método das Três Fases**. Para facilitar a sua compreensão, ilustramos o seu funcionamento através de um exemplo detalhado, no caso o exemplo das Sondas de perfuração de petróleo, cujo enunciado mostramos a seguir.

Uma empresa opera 7 sondas de perfuração de petróleo num campo petrolífero. As sondas trabalham em operação contínua, interrompendo seu funcionamento apenas para manutenção corretiva. O tempo entre falhas é descrito por uma distribuição normal com média 7 e desvio padrão de 1 dia. A manutenção é feita por uma única equipe e sua duração é exponencialmente distribuída com média de 1 dia. Deseja-se simular este problema para avaliar o tempo que as sondas ficam paradas por falta de manutenção. Também se deseja estimar a ocupação média da equipe de manutenção.

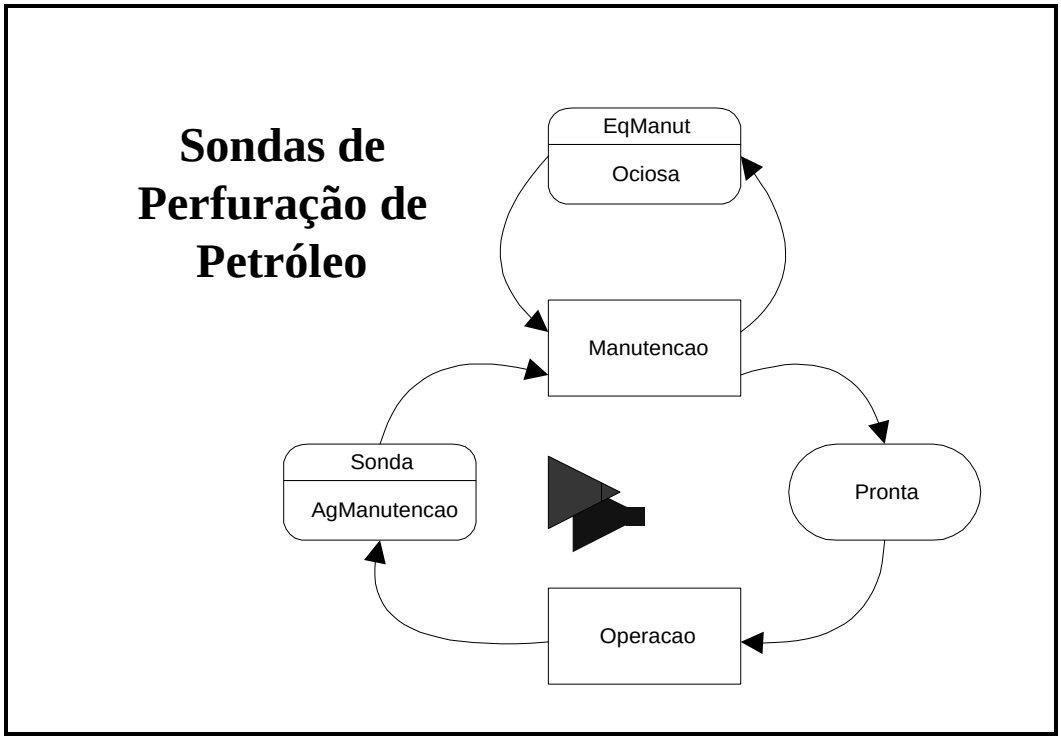


Figura 4.1 Modelo para o problema das Sondas de Perfuração de Petróleo.

Saliby (1995) descreve o Método das Três Fases da seguinte forma (trecho a seguir):

Dentre as alternativas existentes para a estruturação de programas de simulação a eventos discretos, o método das três fases é provavelmente aquele que conduz a programas mais modulares.

Antes da sua descrição, faz-se necessária a definição de dois importantes conceitos: evento e relógio da simulação.

> Evento: corresponde a uma mudança de estado de um sistema que ocorre num ponto bem determinado ao longo do tempo. Por exemplo, quando da chegada de um avião a um aeroporto ou quando do término de um reparo num equipamento. Para efeito de uma simulação, toda atividade é delimitada por dois eventos: seu início e seu término. Reciprocamente, todo evento está associado ao início ou ao término de uma atividade.
>
> Relógio da simulação: indica o particular momento em que se encontra um sistema ao longo de um período simulado. Salvo em situações muito especiais, o relógio numa simulação a eventos discretos é atualizado a intervalos variáveis. O avanço do tempo é determinado pelo evento cuja data programada de ocorrência, dentre todos os eventos já programados, seja mínima. Esta abordagem é conhecida como técnica do próximo evento. O módulo de programa responsável pelo avanço do tempo e pela execução das atividades é denominado executivo da simulação.

4.1 Tipos de eventos

O método das três fases proposto por Tocher (1963), baseia-se no fato de que toda atividade é delimitada por dois eventos: o seu início e o seu término. Além disso, todo início de atividade é visto como um evento condicional. Já o término de uma atividade ocorre numa data pré-programada, uma vez que a duração de uma atividade será sempre conhecida tão logo ela se inicie.

Com base nesta distinção, definem-se dois tipos de eventos associados a uma atividade: eventos B e eventos C.

4.1.1 Eventos Tipo B

Correspondem ao término das atividades e, portanto, têm sua data de ocorrência pré-determinada. Com o objetivo de se ter uma maior modularidade do programa de simulação, define-se um evento B para cada entidade que participa de uma atividade. Por exemplo, no problema da manutenção de sondas, a atividade "Operação" origina um único evento B, correspondente ao término desta atividade para a única entidade que dela participa: a sonda. Já a atividade "Manutenção", da qual participam duas entidades, origina dois eventos B: o término da manutenção para a sonda e o término da manutenção para a equipe.

4.1.2 Eventos Tipo C

Correspondem ao início das atividades e, como tal, têm sua ocorrência condicionada à disponibilidade de entidades nas filas que precedem esta atividade. Ao contrário dos eventos B, define-se um único evento C para cada atividade, independentemente do número de entidades que dela participam. Em outras palavras, o evento C é comum para todas as entidades que participam de uma atividade. Por exemplo, o início da atividade "Operação", no problema das sondas, origina um evento C envolvendo unicamente a entidade "Sonda". Por outro lado, o início da atividade "Manutenção" origina um evento C envolvendo as entidades "Sonda" e "Equipe".

A tabela 4.1, conhecida como tabela de eventos, sintetiza os eventos do problema das sondas. Esta tabela identifica as atividades, entidades envolvidas, eventos C e eventos B (nesta ordem), para o modelo em questão. Sua construção serve de apoio ao processo de modelagem.

Tabela 4.1 Eventos para o problema das sondas

Atividade	Entidade	Identificação	Descrição
Operação		C1	Início de 'Operação'
Manutenção		C2	Início de 'Manutenção'
Operação	Sonda	B1	Término de op. p/ sonda
Manutenção	Sonda	B2	Término manut. p/ sonda
Manutenção	Equipe	B3	Término manut. p/ equipe

As três fases

Executadas sob controle do executivo da simulação, as três fases que caracterizam o método são:

Fase A

Avanço do tempo. Consulta, na lista de eventos B, as datas de término das atividades em andamento. Determina o próximo evento B (o de menor data) e avança o relógio para esta data. Assim sendo, para que o relógio possa avançar, deve-se ter pelo menos uma entrada na lista de eventos B, quando da sua consulta.

Fase B

Execução dos eventos B programados para o momento em que o relógio foi avançado. Neste momento, a atividade correspondente é concluída, liberando as entidades nela envolvidas que são então transferidas para as filas subsequentes.

Fase C

Verificação, para cada uma das atividades, se as condições para o seu início são satisfeitas, ou seja, teste para a execução dos eventos tipo C. Caso passe no teste, a atividade é iniciada: novos eventos B são programados e as entidades que dela participam deixam as filas em que se encontram. Para cada atividade, o teste para seu início é repetido indefinidamente, até que não seja mais possível iniciá-la. A ordem em que o início de cada atividade é testado tem relevância. Esta ordem define a prioridade entre atividades que "competem" por um mesmo recurso (entidade) para o seu início.

Não havendo mais atividades que possam ser iniciadas num particular momento, a fase C é concluída. Em seguida, caso a execução do programa não seja interrompida e desde que a corrida não chegue ao seu final, retorna-se à fase A do processo, repetindo-se indefinidamente o ciclo das três fases.

4.2 Estrutura do programa segundo o método das três fases

A figura 4.1 mostra a estrutura de um programa construído segundo o método das três fases, juntamente com outros elementos da simulação, como a definição do modelo, a inicialização e a emissão de relatório. Observe-se que, antes de passar ao

controle do executivo, é necessário que haja pelo menos uma entrada na lista de eventos B (ou seja, pelo menos uma atividade em andamento), caso contrário o relógio não teria para onde avançar, levando o programa a uma condição de erro. A maneira mais simples e elegante para se resolver este problema consiste em iniciar a simulação com todas as entidades permanentes em filas e, ao término da etapa de inicialização, incluir uma execução da fase C. Com isso, as atividades iniciais são deflagradas.

Além dos eventos B associados às atividades descritas num DCA, costuma-se ter outros dois eventos B especiais, ambos programados durante a etapa de inicialização: um que define o término da corrida; o outro, opcional, que define o término do período de aquecimento. Um período de aquecimento tem por objetivo remover possíveis influências das condições iniciais da corrida no seu resultado final.

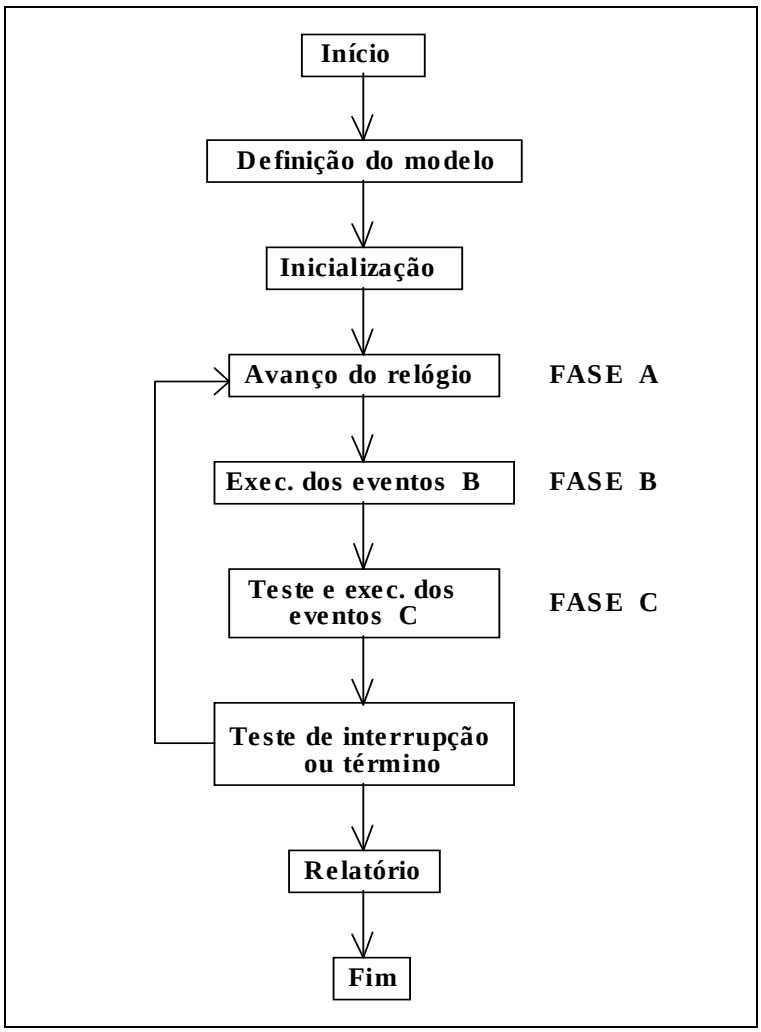


Figura 4.2 Estrutura de um programa de simulação utilizando o Método das três fases.

4.3 Identificação dos eventos B e C numa simulação

A partir do DCA, os eventos B e C são automaticamente definidos. Duas regras simples devem ser observadas:

- a cada bloco de atividade corresponde um único evento C, que caracteriza o seu início;
- a cada bloco de atividade correspondem tantos eventos B quantas forem as entidades que dela participam.

Para uma maior clareza do programa de simulação, é recomendável organizá-lo em função das atividades, com um bloco de instruções para cada uma delas. Este bloco de instruções inicia-se pelo procedimento que define o evento C (início da atividade), seguido da definição dos eventos B correspondentes, com um procedimento para cada entidade participante da atividade. Além disso, como notação, sugere-se identificar os eventos pela letra "C" ou "B", conforme o caso, seguido de um número, tal como usado na tabela 4.1.

4.4 Por dentro do simulador

O processamento da simulação segundo o Método das Três Fases é realizado através do uso de uma estrutura de dados de simulação que é orientada a objetos e com o uso de rotinas construídas especialmente para a simulação a eventos discretos.

No nosso sistema foram utilizadas as classes descritas a seguir. São citados também os objetos utilizados no programa, tendo em vista que a definição da classe não cria o objeto automaticamente, mas somente define um tipo de objeto. O objeto propriamente dito é uma instância de uma classe, que deve ser criado no início do programa. Para cada objeto, os campos fazem o papel de “variáveis” internas e os métodos são como “funções” ou “procedimentos” internos.

A principal classe do sistema é a **TProblema**, que centraliza as informações da simulação como:

- *Titulo*: Título do problema (que aparecerá nos relatórios)
- *Duracao*: Duração das corridas

- *Aquec*: Duração do Aquecimento
- *NCorr*: Número de corridas do problema

Este objeto possui também informações para o controle do andamento da simulação, que são:

- *Temp*: Horário Corrente da simulacao
- *Corrida*: Número da corrida corrente
- *FimCorr*: Horário de término da corrida corrente
- *NumProxB*: Número do próximo evento B
- *NAtiv*: Número total de atvidades no problema
- *NHist*: Número total de histogramas no problema
- *NFila*: Número total de filas no problema

A classe TProblema possui também alguns métodos para o controle da simulação:

- *Constructor Create*: cria um novo objeto da classe TProblema. No nosso caso este objeto é nomeado de **ProbSim**
- *Procedure DefineFimCorrida*: Calcula a variável FimCorr como sendo Duração (Total) + Aquecimento
- *Function Fim* : é do tipo *Boolean* e verifica se já se chegou ao fim da simulação
- *Procedure IncrementaCorrida*: guarda os dados da corrida que está terminando
- *Procedure ProximaCorrida*: inicia uma nova corrida e determina o horário em que deve terminar.
- *Procedure ReinicializaParametros*: prepara os parâmetros gerais para o início da simulação.
- *Procedure ProximoTempo*: Avança o relógio até o tempo relativo ao próximo término de atividade.

Outra classe de extrema importância no sistema é a classe **TEvento_B**, que representa os eventos tipo B do sistema, contendo os seguintes campos:

- *Nome_Atv*: nome da atividade que está para terminar
- *Fila*: fila de destino

- *Nome_Ent*: entidade correspondente
- *Fim_Condicional*: se a atividade tem um fim condicional ou não.

TEvento_B é uma classe. Em tempo de execução, o programa cria vários objetos desta classe, chamandos de Evento_B. Cada objeto Evento_B, por sua vez é colocado em uma lista chamada Lista_Eventos_B.

Há duas variáveis globais do programa associadas aos eventos B. A primeira chama-se Cont_Evento_B e determina o número total de eventos tipo B do sistema. A outra é Maior_Evento_B_Automatico e determina o número de eventos que são criados automaticamente pelo sistema. Mais tarde, o programador poderá criar novos eventos tipo B não automáticos, e utilizará esta variável como base para numerar estes eventos, acrescentando uma unidade ao número indicado.

No sistema há dois tipos de classes relacionadas às atividades do sistema. Existem as atividades gerais (classe **TAtividade**), que são descritas no modelo, como por exemplo “Operação” e “Manutenção”, vistas no problema das sondas. O simulador, na etapa de inicialização cria duas atividades especiais para o problema: *Reinicio e Termino*, a primeira marcando o começo de uma simulação e a segunda marcando o seu fim.

O primeiro tipo de “atividade” é um tipo geral, aplicado somente uma só vez para cada bloco Atividade do diagrama. Há também um segundo tipo de “atividade”, este mais indivualizado para cada entidade que entra na atividade, sendo criado em tempo de execução da simulação, havendo vários para cada tipo de atividade, cada um destes com início, duração e fim específicos. Por exemplo, podemos ter várias “operações” simultâneas envolvendo diferentes “sondas” do sistema e com durações e términos diferentes para cada uma. A este segundo tipo de atividades denominamos como sendo da classe **TAtivExec**, possuindo dois objetos como instâncias desta classe: o *AtvEmPreparo*, utilizado na etapa de preparação das atividades e *AtvCorrente*, que é a atividade que está sendo executada no momento pelo simulador. Note que o simulador, durante a execução do problema, dá atenção somente a uma atividade de cada vez, deixando as outras em suspenso até que chegue a sua vez de ser processada.

Do mesmo modo como ocorre com as atividades do sistema, há também dois tipos associados às entidades do sistema. As entidades gerais, da classe **TEntidade** são definidas na etapa de modelagem e representam tipos gerais para o simulador. Em

tempo de execução, o simulador cria objetos do tipo **TEntidadeAtiv** que contém informações mais ricas. Assim, cada “Sonda” pode ter uma individualidade própria, e o sistema poderá direcioná-las para atividades ou filas diferentes de acordo com o que for preciso. No problema das Sondas, portanto, há somente uma “sonda” da classe TEntidade mas várias “Sondas” da classe TEntidadeAtiv, correspondendo ao número fixado pelo problema.

Há também alguns objetos especiais da classe TEntidade, que na verdade não são entidades do modelo lógico do problema estudado, mas artifícios presentes em todos as vezes que um problema é simulado e que auxiliam o simulador no gerenciamento das informações. São elas *Sistema, EntdoTermino, EntdoReinicio.*

Por fim, temos algumas variáveis globais da simulação que servem como indicadores do estado de execução do problema simulado. São do tipo *Boolean* e possuem os seguintes nomes: Simulacao_Executada, Simulacao_Em_Execucao e Modelo_Checado.

Os códigos das rotinas de simulação para as fases A, B e C podem ser vistos no Apêndice 4.

Capítulo 5
O Ambiente SimVisio

O **SimVisio** é um ambiente integrado de construção e processamento de modelos de simulação que utiliza a metodologia do Diagrama de Ciclo de Atividades. O SimVisio é constituído por um conjunto de Blocos ou *Shapes* de simulação preparados para o uso no programa Visio e por um programa, o SimVisio.Exe que lê as informações do diagrama criado e processa a simulação utilizando as suas informações.

O trabalho de criação do Simvisio é uma evolução de dois sistemas de simulação anteriores: o Simin (Pinto, 1999) e Simul (Saliby, 1995). Historicamente todos estes trabalhos assim como o SimVisio fazem parte de uma família de simuladores que teve início em centros de pesquisa da Inglaterra. O Simul teve como antecessores os sistemas XLSIM, de Carvalho (1975) e o TURBOSIM de Angulo (1983) e de Crookes (1985), todos eles baseados no Turbo Pascal. O trabalho de Crookes foi continuado em duas frentes: uma pelo grupo CASM da London School of Economics (Balmer e Paul, 1986) e outra por Davies e O'Keefe (1989) da Universidade de Southampton. Do trabalho do grupo CASM resultou inicialmente o sistema ELSE (Crookes et Al., 1986) e, posteriormente, o VS6 (Syspack Ltd, 1989), que serviu como referência básica para o Simul. O Simin, em sua versão original, foi o principal resultado da tese de mestrado de Pimentel (1989), do Instituto Militar de Engenharia (IME/RJ), e foi modificado para o paradigma orientado a objetos por Souza (1994). O Simin é baseado no Simul, tendo sido desenvolvido no ambiente Delphi (que é compatível com o Turbo-Pascal) como tese de doutorado de Pinto (1999).

Para criar modelos de simulação no Simvisio devemos primeiramente carregar no Visio com o Painel de simulação de nome DCA e utilizar os Shapes de simulação disponíveis para criar o modelo.

Figura 5.1 Shapes de simulação do Simvisio colocados no Painel DCA do Visio.

O trabalho de programação de um usuário do sistema restringe-se à criação do modelo na interface Visio levando em conta as peculiaridades do problema em estudo.

Este capítulo compõe-se de diversas partes. Inicialmente, são explicados os procedimentos para instalação e testes do SimVisio, seguindo-se de uma visão geral do sistema, mostrando suas diversas características, tais como entrada e saída de dados, telas, recursos disponíveis etc. São mostrados exemplos de uso.

5.1 Instalação do SimVisio

Como o SimVisio funciona sobre o programa Visio de criação de diagramas, e é necessário instalá-lo primeiro. Com o Visio instalado, procede-se com a instalação do SimVisio que é feita através de um programa de próprio (Setup.exe) que basicamente realiza duas operações: primeiramente copia o programa SimVisio.exe no diretório de aplicativos do Visio (\Arquivos de Programas\Visio\Solutions); depois copia os arquivos da Stencil do DCA (DCA.VSS) e os exemplos para o diretório de trabalho (\Arquivos de Programas\Visio\Solutions\Simulacao).

5.2 O DCA - Diagrama de ciclo de atividades

Para trabalhar com o SimVisio é preciso conhecer antes a metodologia que deve ser utilizada para a modelagem dos problemas de simulação. No nosso caso criamos modelos através da metodologia do Diagrama de Ciclo de Atividades ou DCA.

Aproveitamos a explicação apresentada em Saliby (1995) para a montagem de DCA, que foi mostrada a seguir.

Este modelo é, em princípio, uma representação simplificada da realidade. No caso específico da simulação, um modelo é definido por um diagrama lógico descrevendo o comportamento do sistema em estudo. Nosso interesse prende-se à modelagem de sistemas dinâmicos, envolvendo a formação de algum tipo de fila, e cujo comportamento é descrito por uma seqüência de eventos ao longo do tempo. Trata-se da simulação a eventos discretos. Além disso, nossa atenção volta-se, principalmente, para o estudo do comportamento estacionário do sistema".

Há várias abordagens para a modelagem de simulações a eventos discretos, todas elas iniciando-se por algum tipo de representação gráfica do sistema. Descrevemos aqui o uso do diagrama do ciclo de atividades, abordagem padrão da "escola inglesa de simulação", que adotamos como base.

O diagrama do ciclo de atividades (DCA) é uma forma muita prática para se descrever graficamente um problema de simulação. Embora empregue poucos símbolos para esta representação gráfica, o DCA permite a descrição de sistemas relativamente complexos.

O DCA auxilia o analista no entendimento do problema em estudo e facilita a discussão do problema com outros interessados, representando assim um excelente meio de comunicação entre as pessoas envolvidas num estudo de simulação.

São três os principais elementos utilizados num DCA: fontes, atividades e filas. Fonte é a origem de entidades, elemento básico do modelo de simulação, que guarda uma correspondência com as entidades físicas do sistema. Pode-se ter diferentes tipos ou classes de entidades num modelo, sendo que entidades de uma mesma classe têm comportamento similar. Num DCA, descreve-se de forma coletiva o comportamento das entidades de um mesmo tipo ou classe, independentemente da sua quantidade. Assim sendo, a definição de entidades num modelo não é feita de forma individual, mas por grupo de entidades. Por exemplo, na simulação de um bar, como será visto mais adiante, as diferentes classes de entidades consideradas poderiam ser: cliente, garçom e os copos.

Uma entidade é dita permanente quando, durante todo o período simulado, ela está sempre presente no sistema. Por outro lado, uma entidade é dita temporária quando sua permanência é restrita a apenas parte do período simulado. No caso do bar, o cliente seria uma entidade do tipo temporária, enquanto que o garçom e os copos

seriam entidades permanentes. Em termos computacionais, uma entidade permanente tem sua área de memória reservada ao início da simulação. Já uma entidade temporária faz uso da alocação dinâmica, ocupando a memória unicamente durante sua permanência no sistema.

Associado a uma entidade pode ter um ou mais atributos numéricos, atributos estes que podem ter seu valor fixo ou variável ao longo da simulação. Como a definição de entidades é feita por grupos, segue-se que entidades de uma mesma classe são caracterizadas por um mesmo conjunto de atributos. No entanto, os valores destes atributos podem naturalmente variar entre diferentes entidades de uma mesma classe. Voltando ao exemplo do bar, poderia se ter como atributos do cliente a quantidade de copos de cerveja que ele toma, atributo este que seria definido para todos os clientes do bar. Para o garçom, um atributo poderia ser o número de copos que ele serve.

O passo inicial na construção de um DCA consiste na identificação das entidades a serem representadas, seu tipo e atributos. No caso de entidades permanentes, define-se ainda sua quantidade, enquanto que para as entidades temporárias define-se sua fonte ou origem.

Atividade define um estado ativo durante o qual uma ou mais entidades trabalham juntas por um período pré-determinado de tempo. A duração de uma atividade pode ser constante; no entanto, ela é geralmente definida como uma variável aleatória descrita por uma distribuição de probabilidades supostamente conhecida. A duração específica de uma atividade é determinada por um processo de amostragem, quando do início desta atividade. Durante a execução de uma atividade, as entidades nela envolvidas ficam temporariamente indisponíveis.

Num DCA, uma atividade é representada por um retângulo. Cada entidade que dela participa define uma entrada e uma saída do retângulo. A figura 3.1 mostra a representação de uma atividade denominada "Beber", da qual participam duas entidades: "Cliente" e "Copo".

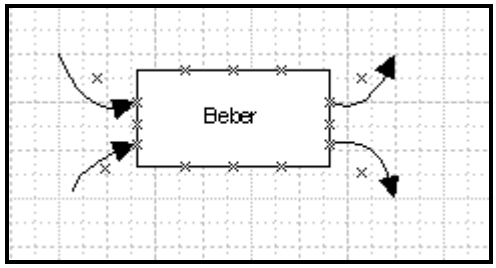


Figura 5.2 Representação da atividade "Beber", envolvendo as entidades "Cliente" e "Copo".

Fila representa o estado passivo, ou de espera, de uma entidade enquanto aguarda condições favoráveis para participar de uma atividade. Uma importante restrição do DCA é que uma fila é exclusiva de uma classe de entidades, não sendo permitida a presença de diferentes tipos de entidade numa mesma fila. Outra importante diferença em relação a uma atividade é que o tempo de permanência numa fila não é pré-determinado; somente com a execução da simulação é que este tempo ficará determinado. Num DCA, é obrigatória a alternância entre atividades e filas. Há casos em que esta regra torna necessária a definição de filas artificiais ou "dummy", nas quais a espera de fato não ocorre.

Num DCA, uma fila é representada por um círculo. A figura 3.2 mostra a representação de uma fila denominada "Cheio", exclusiva da entidade "Copo".

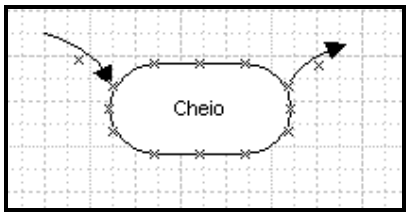


Figura 5.3 Representação da fila "Cheio", exclusiva da entidade "Copo".

Após a identificação das entidades, constrói-se o diagrama do ciclo de vida para cada classe de entidade. Um diagrama do ciclo de vida descreve as várias etapas percorridas por uma entidade durante a operação do sistema. Ele é composto de uma seqüência alternada de atividades e filas.

A seguir enunciamos algumas regras básicas para a construção de DCAs. A observância destas regras serve para uma boa compreensão da lógica do problema pelo usuário e facilita o SimVisio na montagem correta do modelo computacional que realizará a simulação propriamente dita. Eis as regras de construção do DCA.

1. As filas são de uso exclusivo de uma classe de entidades; não misturar entidades diferentes numa mesma fila.
2. Todos os conectores devem estar conectados nas duas pontas.
3. Um bloco não deve estar conectado a si mesmo.

4. Deve-se alternar filas e atividade em um ciclo, ou seja, não deve haver duas filas em seqüência nem duas atividades em seqüência. Entre uma atividade e uma fila podem, entanto, haver blocos de decisão especiais como Condição, Inspeção e Auto (Busca automática de menor Fila). Se necessário, usar filas "Dummy" para preencher o espaço entre duas atividades.
5. Cada atividade deve ser precedida por Fila, Fonte ou FilaRec.
6. Em uma atividade, o número de conectores que entram e que saem é o mesmo.
7. Deve-se fechar os ciclos das entidades, quer sejam permanentes (iniciam com uma fonte) quer sejam temporárias (iniciam com uma FilaRec). Os diversos ciclos devem terminar em uma Fonte ou FilaRec.
8. Todos conectores devem fazer parte de um ciclo de entidades ou recursos.
9. Indicar corretamente o ciclo de cada entidade utilizando pontos simétricos em cada bloco de atividade. O SimVisio inicia com todos os arcos com a cor preta e separa automaticamente os ciclos com cores diferentes para cada entidade pelo modo como são construídos os ciclos.
10. Quando vários ciclos passam por uma atividade deve-se, sempre que possível, percorrer atividades num mesmo sentido. Note que entidades diferentes só podem interagir em atividades.
11. Indicar prioridades, condições de início, duração de atividades e disciplinas de filas não FIFO.

A observância destas regras leva à construção de modelos com uma lógica clara. Descrevemos a seguir os blocos para a construção dos DCA que são utilizados no ambiente SimVisio.

5.3 Blocos do DCA no SimVisio

Para criar um modelo de simulação basta abrir uma nova página do Visio em branco e ir arrastando os elementos desejados a partir do painel do DCA para a página em branco. Para mudar o nome de um elemento na página basta clicar sobre o mesmo, esperar que apareça a caixa de edição e editar o texto sobre o Shape. Cada elemento tem também algumas propriedades, que aparecem quando se clica com o botão da direita sobre o mesmo e se escolhe a opção "Properties".

Para ilustrar o uso do SimVisio iniciaremos com um exemplo simples, um sistema que simula um caixa de **Banco**. Neste sistema, o intervalo entre chegadas de dois clientes consecutivos segue uma distribuição de probabilidade Exponencial Negativa com média 1 minuto, possui fila única e um único caixa com tempo médio de atendimento que segue uma distribuição exponencial negativa com média 0,8 minuto. Utilizando a notação da Teoria das Filas classificamos este caso como um típico sistema M/M/1, onde o primeiro M indica o tipo de chegada, no caso exponencial ou "Markoviano", um atendimento também do tipo "Markoviano" e número de servidores igual a um. O modelo DCA construído no SimVisio do problema M/M/1 tem a seguinte forma:

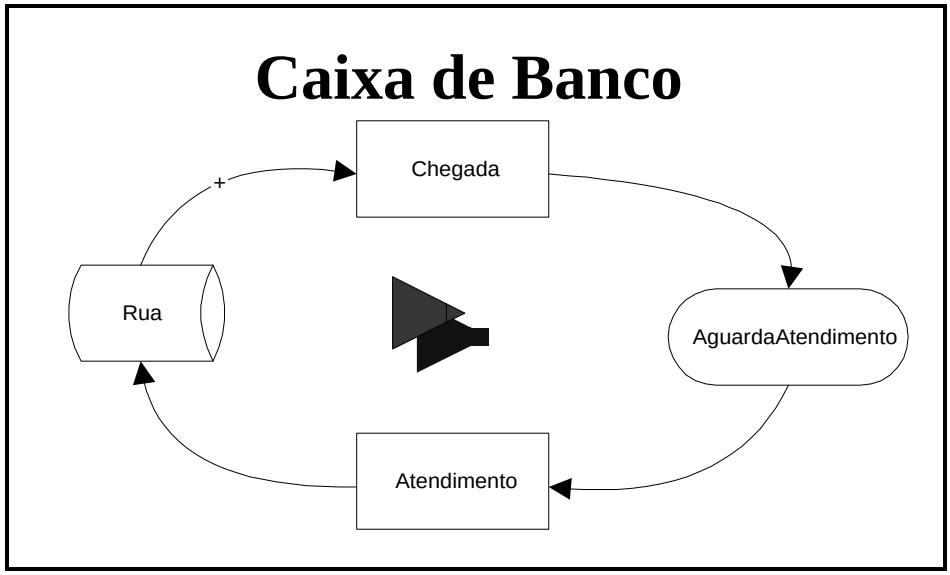


Figura 5.4 Modelo DCA feito no SimVisio do problema do banco (M/M/1)

Para criar este modelo em SimVisio é bem simples.

1º. Passo) Abra o Visio

2º. Passo) Inicie um modelo em branco (escolha "Blank Drawing.vst" ou indique no menu principal File/New e depois "Drawing").

3º. Passo) Abra a *stencil* de simulação denominada DCA.vss. No menu: File/Stencils/Simulacao/DCA. Se quiser, pode utilizar o ícone de abertura de stencil (Pasta que se abre com uma folha verde dentro).

4º. Passo) Arraste e solte um *shape* do tipo "Fonte" a partir da *Stencil* DCA para a folha em branco.

5º. Passo) Faça o mesmo com mais dois shapes do tipo “Atividade”, um shape do tipo Fila, e um shape “Simulação”, dispondo conforme a figura acima.

6º. Passo) Una os shapes através de um “Conector”. Para tanto basta arrastar o mesmo a partir da stencil DCA e colocar no modelo, tomando o cuidado de fazer com que as extremidades dos conectores coincidam com os pontos de conexão dos Shapes que ele deve ligar. Note que os pontos de conexão estão destacados com marcas de “X” de cor azul e que, quando a extremidade de um conector toca um ponto de conexão de um determinado shape, ela se torna vermelha indicando que a conexão é válida. As extremidades dos shapes que são pontiagudas indicam a direção que a entidade deve percorrer durante a simulação; siga o exemplo da figura. Um cuidado importante é o de colar os conectores nos pontos certos dos shapes; a um ponto de conexão de entrada há um e somente um correspondente ponto de saída. A figura abaixo indica as correspondências para um shape Atividade. No caso, A1 corresponde a A2, B1 a B2 e assim por diante.

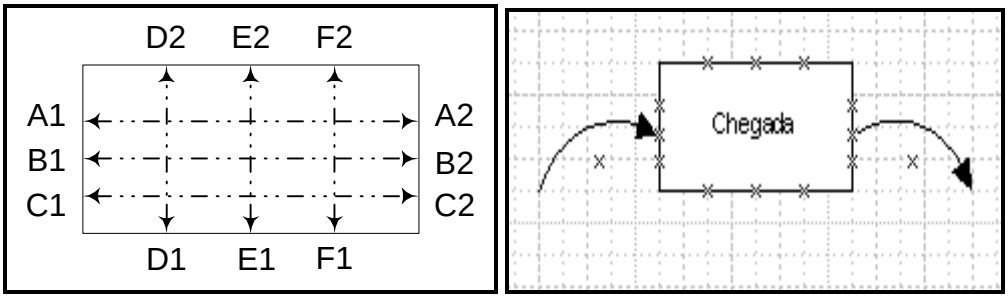


Figura 5.5 Correspondência entre pontos de conexão de entrada e de saída em um shape tipo Atividade. (A) O ponto A1 corresponde a A2, o B1 ao B2 e assim por diante. (B) Exemplo de conectores corretamente posicionados para a atividade “Chegada”.

7º. Passo) Agora, para preparar o modelo de modo a que se possa rodar a sua simulação, é necessário indicar corretamente as propriedades de cada elemento do modelo. Para tanto vamos ver analisar cada shape em particular, e preencher as propriedades uma a uma. Para ter acesso às propriedades de um elemento do modelo, basta dar um clique com o botão direito do mouse sobre o mesmo e selecionar “Properties”. A descrição de cada bloco de simulação é feita a seguir.

5.4 Descrição Dos Blocos Principais

1º BLOCO: FONTE

Representa a porta de entrada de entidades temporárias no sistema. As entidades temporárias são aquelas que, após o atendimento, deixam o sistema.

Exemplo: Em um modelo representativo de um caixa de banco, pode-se criar uma Fonte de nome Rua para representar a chegada de clientes ao banco. Neste caso, os clientes são entidades temporárias, pois deixam o sistema após o atendimento. Esta denominação foi criada para diferenciar de entidades permanentes, que são aquelas que estão ativas no sistema durante todo o período de simulação. Vemos na figura abaixo um exemplo de utilização do elemento Fonte para o exemplo do Banco.

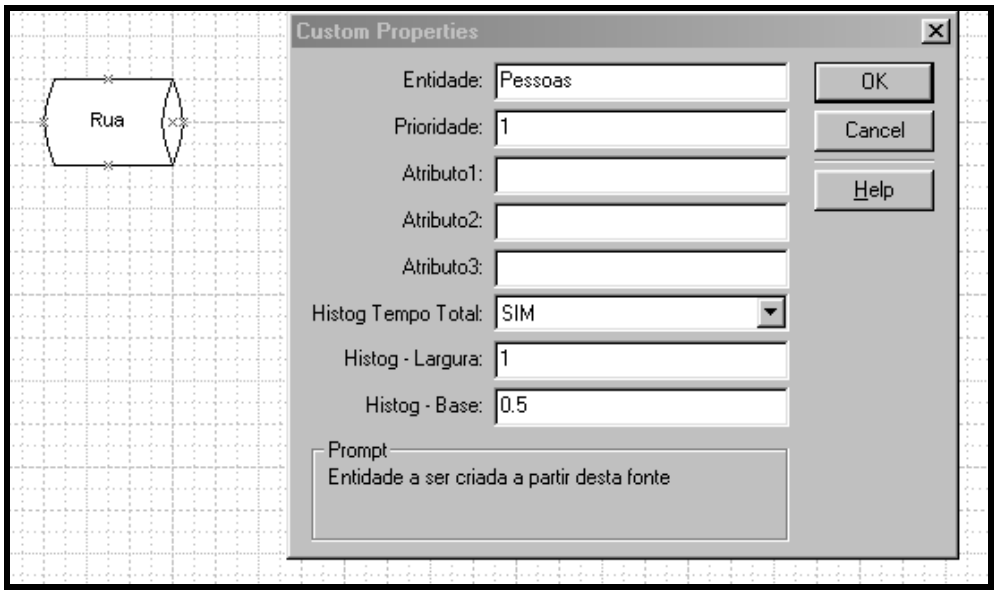


Figura 5.6 Fonte "RUA" utilizada no exemplo do Banco, onde se indica a entrada de entidades da classe "Cliente".

Propriedades:

Entidade: nome da entidade a ser criada a partir desta fonte.

Prioridade: prioridade das entidades desta fonte frente a outras entidades, quando disputam quais serão atendidas primeiramente em uma atividade em que concorrem.

Atributo: nomes de atributos associados à entidade determinada acima. Para indicar mais de um, separar por "&".

Histograma Tempo Total: Determina se deve ser criado ou não o histograma de tempo total da entidade no sistema, que pode ser visto depois de terminada a simulação.
Histograma – Largura: Largura de cada faixa do histograma acima.
Histograma – Base: Início da primeira faixa de histograma. (P.ex. Se Base = 0.5 e Largura =3, então a primeira faixa é de 0.5 a 3.5, a segunda de 3.5 a 6.5 e assim por diante)

2º BLOCO: ATIVIDADE

Representa as atividades do sistema. Várias entidades podem participar da atividade. Uma atividade somente inicia quando houver entidades disponíveis nas filas que a precedem. A duração da atividade possui uma distribuição de probabilidade que deve ser indicada quando a atividade é criada. Ao final da atividade podem ser realizados cálculos com atributos das entidades envolvidas.

Exemplo: No banco há um caixa que realiza a tarefa de atendimento segundo uma distribuição de probabilidade Exponencial Negativa de Média 0,8 e capacidade igual a 1. Para a atividade de chegada, o intervalo entre chegadas tem distribuição Exponencial Negativa com Média 1 e capacidade Infinita (por definição as atividades de chegada devem ter capacidade infinita).

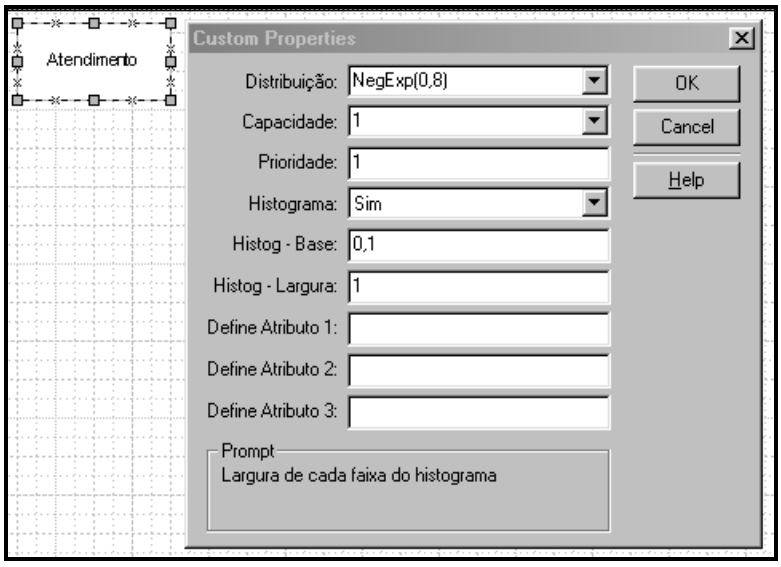


Figura 5.7 Atividade "Atendimento" utilizada no exemplo do banco, onde se indica distribuição de probabilidade da duração da atividade e a capacidade de atendimento.

Propriedades:

Distribuição: Distribuição de Probabilidade da duração da atividade. Pode ser simples ou uma composição de várias distribuições e/ou atributos. Os tipos estão indicados na própria lista de opções:

- Normal
- Exponencial Negativa (NegExp)
- Weibull
- Poisson
- Erlang
- Triangular
- Uniforme
- Inteira Uniforme (IntUniforme)
- Lognormal
- Gama
- Beta.

Cada uma possui um conjunto de parâmetros próprios que devem ser preenchidos, por exemplo, NegExp(0,8). Os números devem ser separados por ":".

Capacidade: Número de atendimentos que podem ser realizados ao mesmo tempo. Se a capacidade for infinita as entidades passam pela atividade sem criar nenhum tipo de fila. Indicar um número N para a capacidade equivale a dizer que há N servidores atendendo esta atividade, por exemplo, um caixa no atendimento do banco. Note que as atividades de chegada, que estão imediatamente após uma fonte, devem ter, por definição, capacidade infinita.

Prioridade: Indica o critério para determinar qual atividade atenderá primeiro quando duas atividades estiverem disputando a mesma entidade para iniciar o atendimento. A maior prioridade é 1, e as outras são maiores que 1.

Histograma: Neste campo pode-se solicitar que seja feito o histograma dos tempos de duração da atividade durante o período de simulação.

Histog – Base: Base do histograma de tempo de atividade, ou seja, o início do primeiro intervalo a ser contado.

Histog – Largura: Largura dos intervalos do histograma de tempo de duração da atividade. Por exemplo, Se base=0,5 e largura=1, então o 1^{o}. intervalo vai de 0,5 a 1,5; o segundo de 1,5 a 2,5 e assim por diante até o 15^{o}. intervalo. Já o 16^{o}. intervalo vai de 15,5 ao infinito.

Define Atributo (1, 2 e 3): Neste campo podem ser atribuídos novos valores aos atributos da entidade que participam da atividade. Esta atribuição é feita somente no final da atividade, e não em seu início. Por exemplo, se na respectiva fonte foi criado o atributo Tarefas para a entidade Pessoas, podemos indicar neste campo que Tarefas=IntUniforme (1:5), sendo que a cada entidade do tipo pessoas que passar pela atividade, o seu atributo tarefas deve receber um valor aleatório de 1 a 5 uniformemente distribuído. Note que podem ser feitas no máximo 3 atribuições por atividade.

3º BLOCO: FILA

Representa as filas do sistema. As filas estão localizadas antes de cada atividade, sendo que deve ser indicada uma fila para cada tipo de entidade que entra nas atividades. As filas podem ter capacidade infinita ou limitada. Quando se atinge o limite de uma fila, pode haver dois tipos de reação: ou se eliminam as novas entidades que tentam entrar na fila, ou se bloqueia a atividade anterior para que ela não envie novas entidades para esta fila. Não é necessário indicar a entidade que irá permanecer na fila; a identificação é feita automaticamente pelo programa quando é analisado o diagrama.

Exemplo: No caso do banco, devemos colocar uma fila antes da atividade de atendimento. Note que o nome da fila foi colocado sem espaço em branco entre “Aguarda” e “Atendimento” por restrições do programa, que não aceita este tipo de caracteres.

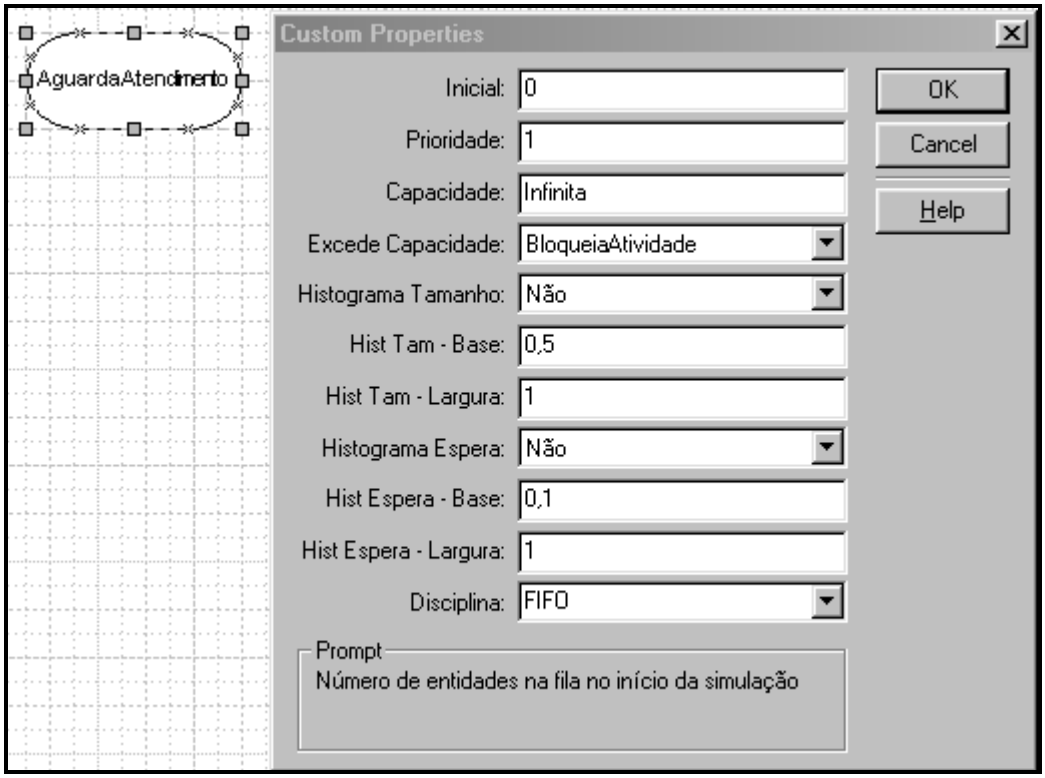


Figura 5.8 Fila "AguardaAtendimento" utilizada no exemplo do banco.

Propriedades:

Prioridade: representa a prioridade desta fila sobre as outras filas, quando uma atividade tem de escolher entre duas filas que fornecem o mesmo tipo de entidades. A maior prioridade é um e as outras são maiores que 1.

Capacidade: indica o tamanho máximo da fila, acima do qual o programa deve agir segundo o indicado no item abaixo: Excede Capacidade.

Excede Capacidade: quando a capacidade da fila é atingida, ou a entidade que chega é eliminada do sistema (SairDoSistema) ou a atividade anterior é bloqueada (BloqueiaAtividade).

4º BLOCO: SIMULAÇÃO

Determina os parâmetros da execução da simulação, como tamanho e número de corridas e tamanho do período de aquecimento. Define-se o nome do problema e as informações que se desejam ver durante a simulação como, por exemplo, a atividade que está sendo processada no momento.

Exemplo: No caso do Banco, o título do problema é “CaixaDeBanco”, a duração da corrida, o tamanho do período de aquecimento e o número de corridas está indicado como na figura. Foram solicitadas informações para visualizar o andamento da simulação, mas não passo a passo.

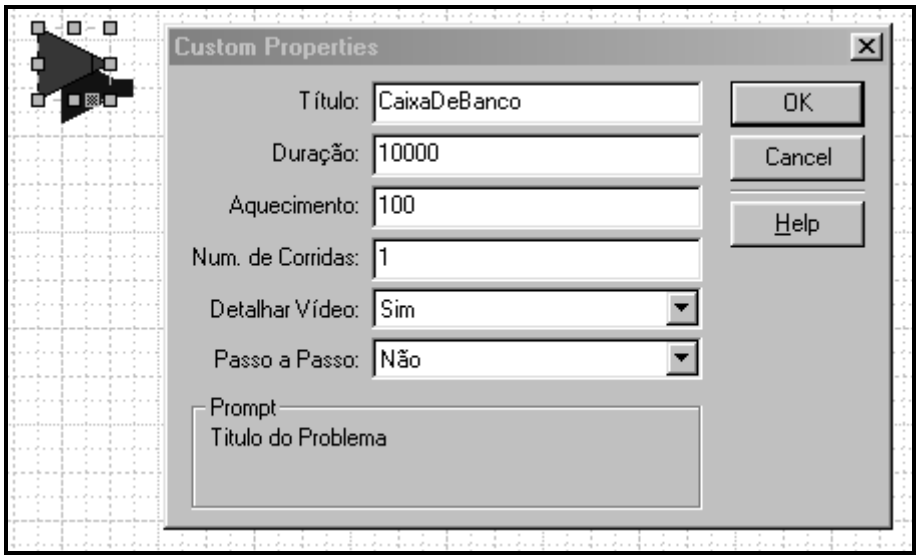


Figura 5.9 Propriedades da simulação, para o exemplo do banco, onde se indica o tamanho da corrida de simulação e nome do problema “CaixaDeBanco”

Propriedades:

Nome do problema: Identificador do problema a ser estudado, e que aparecerá no cabeçalho dos relatórios. O nome não pode conter espaços em branco; para nomes compostos, utilizar maiúsculas no início de cada palavra como, por exemplo, CaixaDeBanco. Não se deve utilizar caracteres pouco comuns, como por exemplo “ç” ou acentos de qualquer gênero: “á’”, “à”, “ü” ou “â”.

Tamanho da corrida: Tamanho do período de cada corrida de simulação. Deve-se usar a mesma unidade de tempo (horas, minutos, segundos ou dias) em todos os parâmetros de tempo do modelo, como por exemplo, na determinação do tempo de duração da atividade. Não é necessário indicar a unidade utilizada no preenchimento deste campo. Não se deve colocar pontos para separar milhares, como por exemplo “10.000”.

Aquecimento: Período em que a simulação ocorre, mas em que não são contadas as estatísticas, como tamanho da fila, duração da atividade e tempo de espera na fila. Ao final do aquecimento, o relógio continua a contagem sem que zere o marcador.

Detalhar Vídeo: Solicita que seja mostrado em uma janela o andamento da simulação com informações do relógio e da atividade que está sendo executada.

Passo a Passo: Caso esteja indicado “Sim”, o simulador irá parar a cada atividade que é iniciada, somente continuando se o usuário clicar o botão para continuar a simulação.

Agora que foram explicados os detalhes dos primeiros quatro blocos principais, já é possível rodar a simulação. Para tanto é preciso chamar o programa que contém as rotinas de leitura do modelo e execução da simulação. Este programa é o SIMVISIO.EXE e está localizado no diretório “Solutions” do Visio. O Visio, por sua vez, costuma estar instalado no diretório “Arquivos de programas” do Windows. O programa SimVisio.exe pode ser chamado pelo Menu Tools/Macros/Macro e escolhendo Simvisio dentre as opções apresentadas. O mais prático, no entanto, é fazer o que está indicado no quadro a seguir.

Associe o evento “duplo clique” à execução do SimVisio. Para tanto basta clicar com o botão da direita sobre o ícone da Simulação e escolher Format/Double Click/Run Macro e escolher Simvisio na lista correspondente. Assim, todas as vezes que se der um clique duplo sobre o ícone de simulação, o Simvisio.exe rodará automaticamente.

Ao rodarmos o modelo do Banco, aparece em primeiro lugar uma janela onde são mostradas as etapas de análise do modelo. Nesta etapa o Simvisio lê as propriedades do diagrama DCA e cria os componentes de simulação internos, para que ele possa rodar o algoritmo de simulação sobre estes componentes. É feita uma verificação dos erros de sintaxe do modelo e por fim aparece a janela de execução da simulação.

5.5 Tela Principal do Programa SimVisio

Ao fim da simulação aparece a mensagem “Simulação terminada com sucesso”. Clicando na caixa de mensagem, aparece a tela principal do SimVisio para o usuário. Nela pode-se escolher os relatórios e histogramas do problema.

Figura 5.10 Tela Principal do SimVisio.

5.6 Tela de Relatórios das Atividades, Entidades e Histogramas

Para solicitar o relatório de entidade podemos utilizar o menu ou clicar diretamente sobre o ícone correspondente. Há dois tipos de opções de relatórios: o Relatório de Atividades e Entidades e o Relatório de Histogramas. O primeiro contém, além dos dados de atividades e entidades, os dados referentes ao número de entidades que passaram em cada fila bem como as estatísticas dos blocos de Inspeção e Condição. A seguir podemos ver o Relatório de Atividades e Entidades para o problema do Banco. Note que a taxa de utilização verificada está bem próxima do que a Teoria das Filas determina: 0,8/1,0 = 80%.

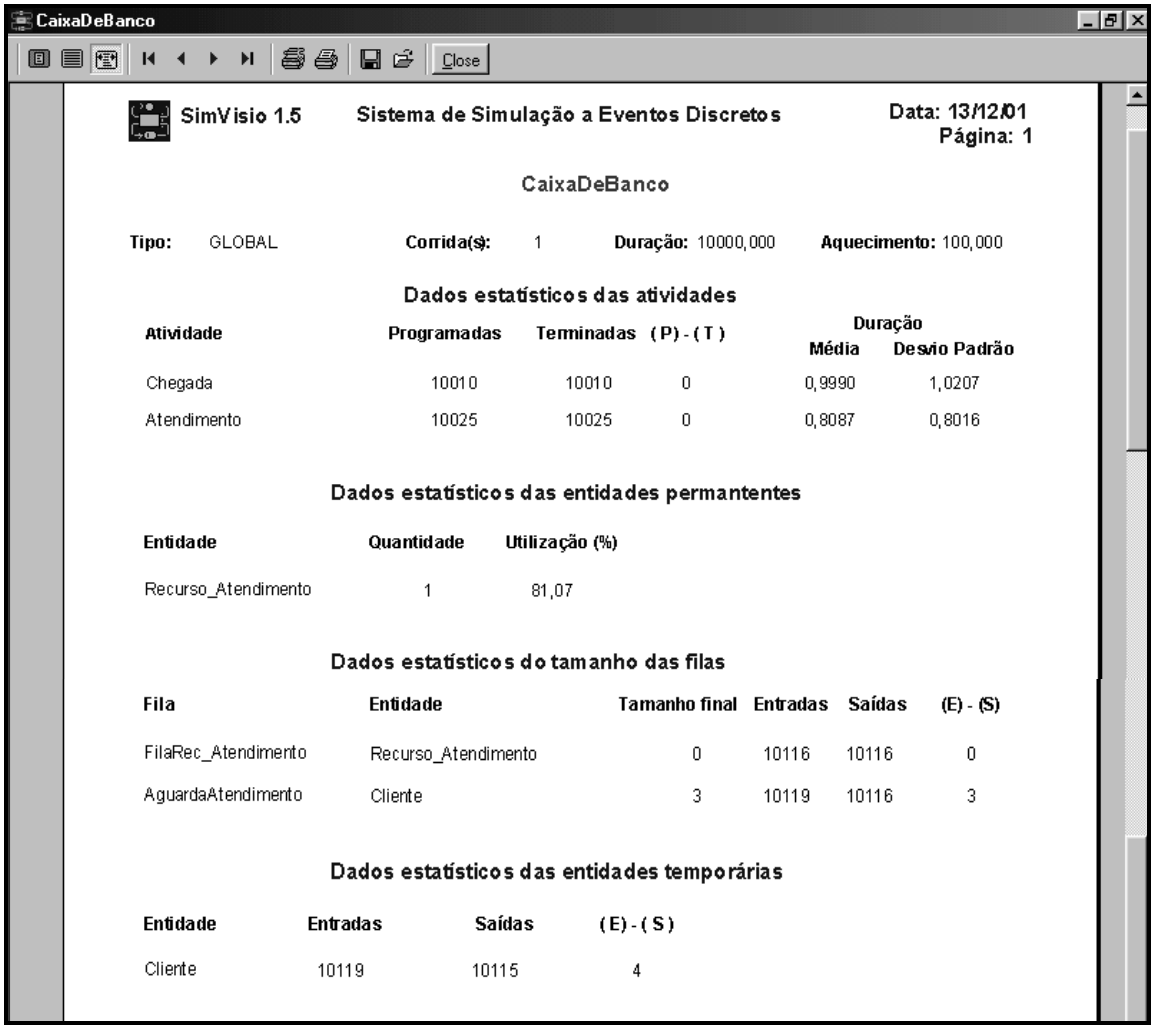


Figura 5.11 Relatório de Atividades e Entidades para o problema do Banco.

Neste relatório aparece um Recurso de nome “Recurso_Atendimento” que foi criado automaticamente pelo SimVisio para servir à Atividade “Atendimento”. Os

recursos são entidades permanentes do sistema e o seu nome sempre é criado pelo SimVisio acrescentando-se “Recurso_” ao nome da atividade a que serve. Do mesmo modo foi criada automaticamente também a fila “FilaRec_Atendimento”.

5.7 Histogramas

A seguir mostramos as telas correspondentes aos histogramas de Tempo de Espera e de Tamanho da fila, solicitados no Menu Gráficos/Histogramas do SimVisio. Ambos histogramas foram solicitados quando foram definidas as propriedades da Fila “AguardaAtendimento”.

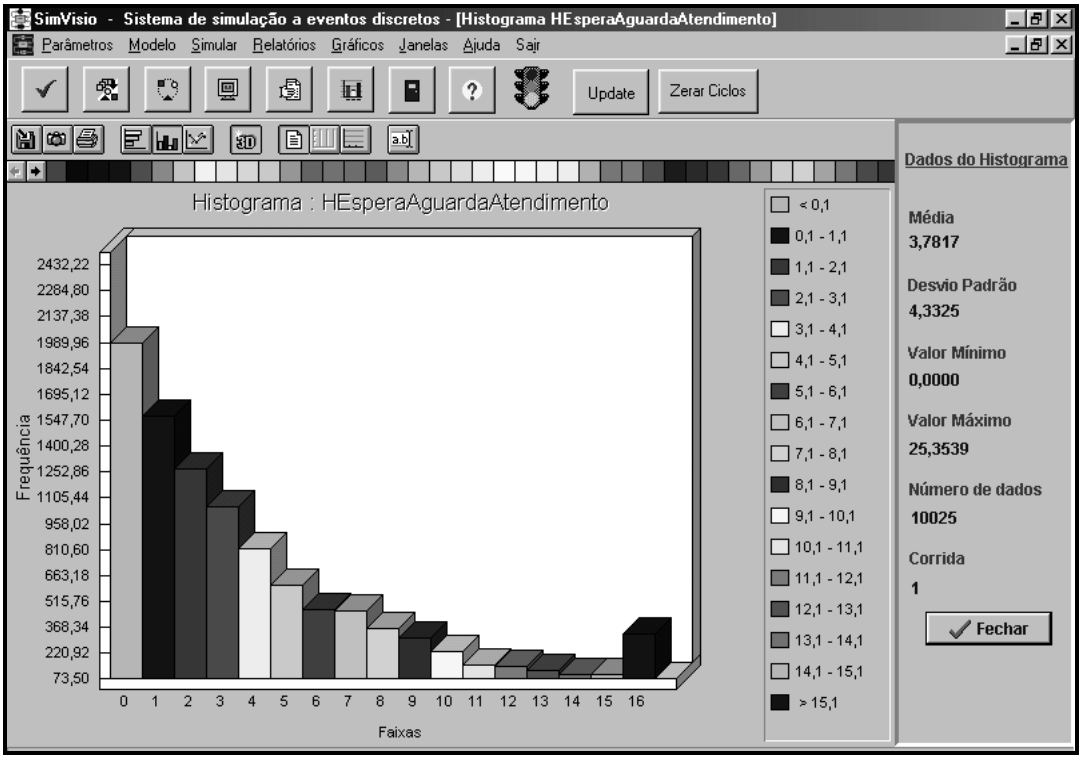


Figura 5.12 Histograma de Tempo de Espera na Fila “AguardaAtendimento” para o problema do Banco.

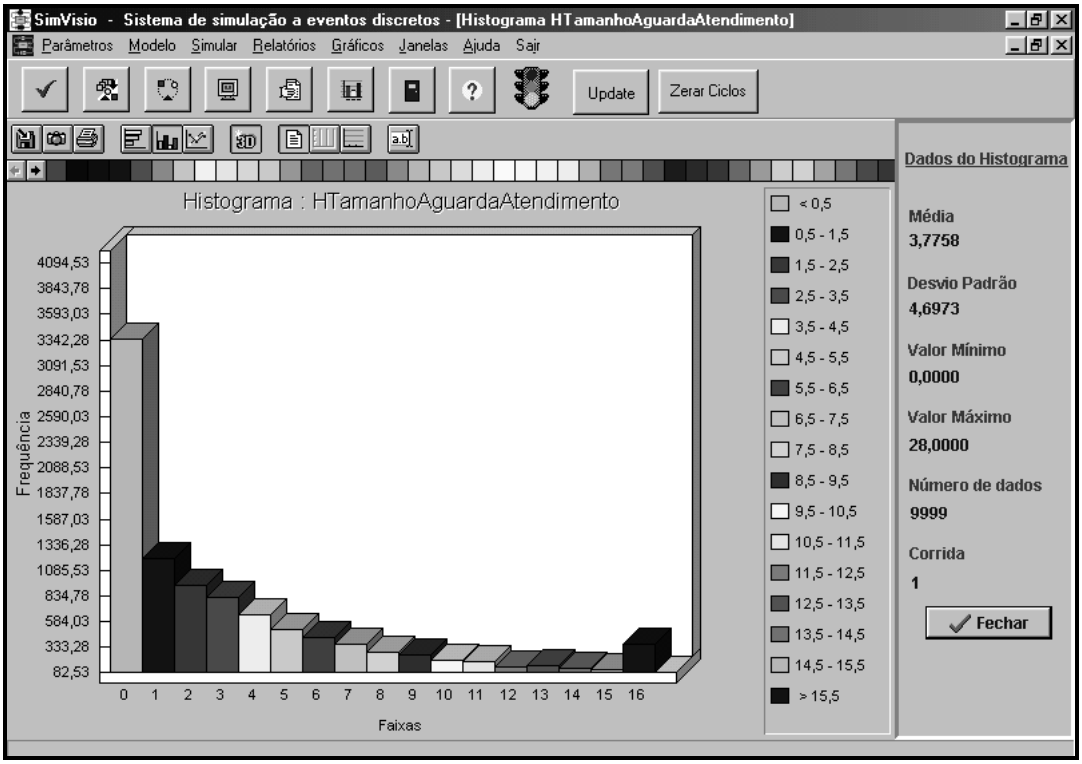


Figura 5.13 Histograma de Tamanho da Fila "AguardaAtendimento" para o problema do Banco.

5.8 Arquivos de dados dos histogramas

Uma funcionalidade muito importante que foi colocada à disposição do usuário foi a possibilidade de exportação dos dados de cada histograma para um arquivo texto. Estes arquivos são solicitados pelo usuário quando é definida a propriedade Histograma dos blocos Fila, FilaRec e Atividade. Há nestes blocos três opções de escolha: "Não" para indicar que não se desejam histogramas, "Sim" para a criação de histogramas simples sem a geração de arquivos texto com os dados e "ArquivoTxt" que indica que se deseja um arquivo em formato TXT para todos dos dados do histograma.

Caso seja solicitado algum arquivo texto o programa automaticamente cria um sub-diretório com o mesmo nome que o do documento do modelo em formato Visio (.VSD). Assim, por exemplo, se o problema em questão é o Sondas.vsd, cria-se no diretório ao qual pertence o arquivo Sondas.vsd um novo sub-diretório de nome "Sondas". Para cada histograma para o quail foram pedidos os seus dados são criados vários arquivos contendo o nome do histograma e o número da corrida. Assim, por exemplo, no problema das Sondas, se quisermos os dados do Histograma da duração da atividade Manutencao, teremos os seguintes arquivos: HManutencao_1.txt para a primeira corrida, HManutencao_2.txt para a segunda corrida e assim por diante.

Com base nos dados contidos nestes arquivos é possível fazer uma série de análises estatísticas através de pacotes próprios para este fim.

Cabe ressaltar que a solicitação de dados dos histogramas no formato texto deixa o processamento da simulação mais lenta, de modo que devem ser criteriosamente escolhidos para exportação somente aqueles dados que forem realmente importantes.

5.9 Recursos Adicionais Do SimVisio

Há outros blocos disponíveis no SimVisio, e o segundo exemplo utiliza alguns destes recursos. No **modelo do Bar**, os clientes chegam da rua para tomar cerveja, numa quantidade que varia aleatoriamente em função da sede de cada um. Os intervalos entre chegadas consecutivas são exponencialmente distribuídos com média de 10 minutos. A quantidade de copos que cada cliente toma é definida quando da sua chegada, através do atributo Sede. A Sede de um cliente varia de acordo com uma distribuição inteira uniforme com um mínimo de 1 e um máximo de 4 copos. Chegando ao bar, um cliente aguardará sua vez de ser servido. Uma vez terminada a atividade de Servir, cuja duração segue uma distribuição normal com média de 6 minutos e desvio padrão de 1 minuto, o cliente beberá seu copo a seguir. O tempo para beber um copo distribui-se uniformemente com valores inteiros entre 5 e 8 minutos. Este ciclo irá se repetir até que o cliente tenha sua sede saciada. Dois garçons são responsáveis pelo atendimento dos clientes e pela lavagem dos copos usados. O atendimento, além do cliente, exige também que um copo limpo esteja disponível. A lavagem dos copos tem duração constante e igual a 5 minutos, podendo ser lavados até 3 copos de uma só vez. Supõe-se ainda que o bar dispõe de 70 copos.

O modelo em DCA do problema do Bar tem a seguinte configuração (figura 5.14).

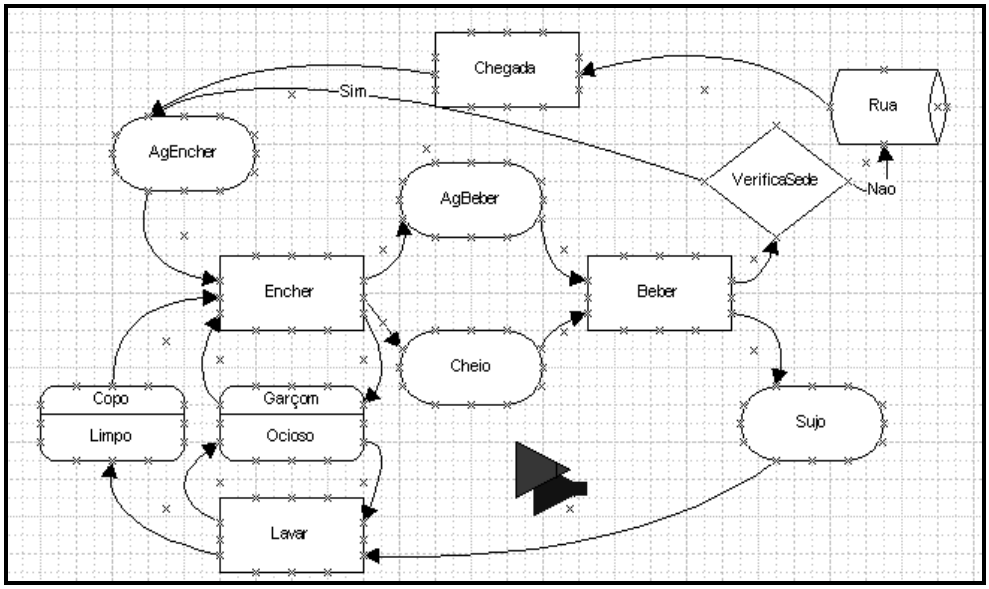


Figura 5.14 Modelo em DCA do problema do Bar.

Para modelar o exemplo do Bar é necessário utilizar alguns blocos que já foram apresentados anteriormente, como Fonte, Fila, Atividade e Simulação. Além destes, são também necessários os blocos FilaRec e Condição, que são descritos a seguir.

5º BLOCO: FILAREC

Um bloco FilaRec se assemelha a um bloco Fila, mas com algumas propriedades adicionais. Quando se coloca um bloco deste tipo no modelo, o simulador se encarrega de preencher a fila com entidades permanentes no início da simulação. O nome da entidade permanente, sua quantidade e seus atributos são definidos na própria janela de propriedades da FilaRec. Depois de iniciada a simulação, os bloco FilaRec se comporta como um bloco Fila simples, segundo as suas características de capacidade, disciplina de retirada e histogramas. Pode-se dizer, portanto, que um bloco FilaRec é como uma combinação de um bloco Fila com um bloco Fonte, mas com a diferença que o FilaRec cria entidades permanentes e o Fonte entidades temporárias.

Exemplo: No exemplo do Bar, é criada uma FilaRec para o ciclo da entidade Garçom e também para o ciclo da entidade Copo.

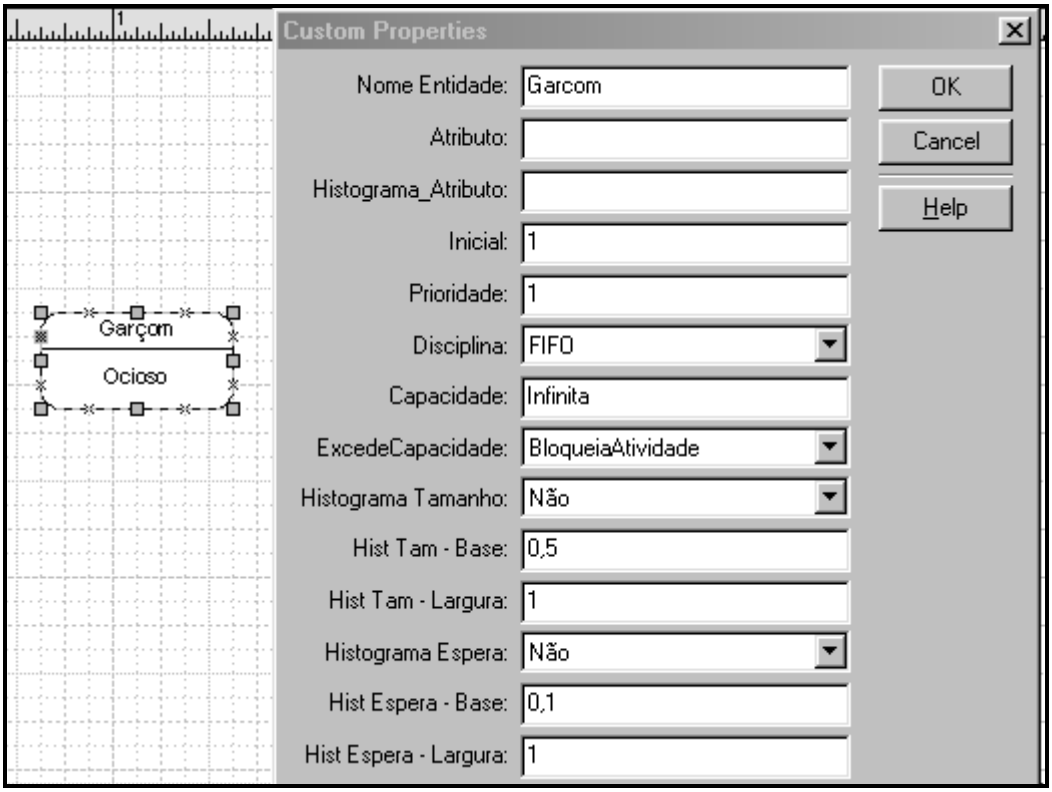


Figura 5.15 FilaRec "Ocioso" utilizada no exemplo do Bar, onde se indica a utilização de entidades da classe "Garcom".

Propriedades:

Nome Entidade, Atributo e Histograma Atributo: mesmas características que no bloco Fonte.

Inicial, Prioridade, Capacidade, Excede Capacidade, Histograma Tamanho e Histograma Espera: mesmas características que no bloco Fila.

6º BLOCO: CONDIÇÃO (Desvios condicionais), CONECTORES SIM e NÃO.

O bloco de condição estabelece desvios condicionais para as entidades que saem de um bloco Atividade. É determinado um teste para a entidade que termina a atividade, envolvendo um atributo desta entidade, o operador matemático para o teste (<,>, <=, >= ou =) e o valor limite para o teste. Caso o atributo em questão esteja de acordo com a condição de teste, a entidade é encaminhada para a fila para a qual o conector "Sim" indicar; caso contrário, encaminha-se para onde o conector "Não" estiver apontando.

Exemplo: No exemplo do Bar, a Condição verifica se o atributo “Sede” da entidade “Cliente” possui valor “>” a “0” (zero). Caso isto seja verdadeiro, ou seja, caso o cliente ainda tenha sede, volta para a fila para aguardar encher o copo novamente. Caso contrário, ou seja, se não tem mais sede, o cliente vai para o fim do ciclo (saída).

Figura 5.16 Condição “VerificaSede” utilizada no exemplo do Bar, onde se indica o atributo a ser verificado (Sede), a condição (>) e o valor de comparação (0).

Propriedades:

NomeAtributo: Nome do atributo cujo valor vai ser avaliado. Note que o nome do atributo deve estar previamente definido na respectiva Fonte ou FilaRec, no momento da criação da entidade a que pertence.

Condição: Pode ser “>”, “<”, “<=”, “>=”, “<>” ou “=”.

Valor: Valor de comparação. Deve ser um número real sem separação de casas de milhar por “.” (ponto) ou “,” (vírgula).

7º BLOCO: INSPEÇÃO

O bloco de inspeção deve ser colocado sempre após um bloco de atividade, e serve para direcionar a entidade ao final de sua execução. Define-se a porcentagem de entidades a serem aprovadas, havendo várias porcentagens de aprovação. As restantes serão, portanto, rejeitadas, somando entre aprovadas e rejeitadas o total de 100%. As rejeitadas irão para a fila indicada pelo conector “Sim” (sem defeito) e as aceitas para a fila indicada pelo conector “Não” (com defeito). As entidades rejeitadas podem voltar para uma parte anterior do ciclo, para serem reprocessadas. Para exemplificar o uso do bloco inspeção, tomemos o exemplo de uma máquina em operação em uma fábrica, onde há 10% de probabilidade de produzir uma peça defeituosa. Caso a peça tenha

defeito, é processada novamente na máquina. Caso contrário, segue na linha de produção.

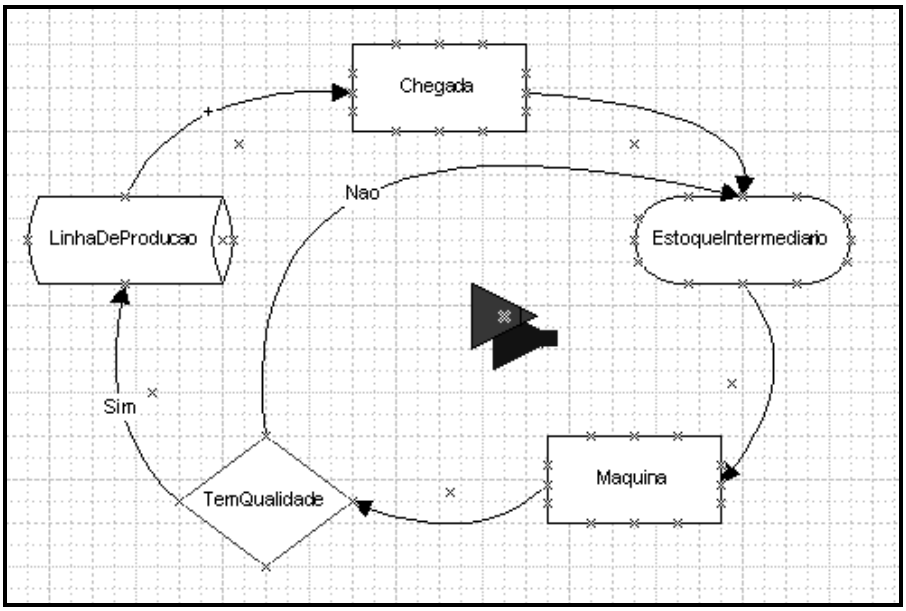


Figura 5.17 Exemplo de uma Fábrica com uma máquina, onde se tem uma Inspeção. As peças ruins são processadas novamente e as boas seguem adiante.

Exemplo: Em um modelo que simula a fábrica, deve-se indicar a porcentagem de peças boas (90%). As peças boas seguirão para a fila indicada pelo conector “Sim” e as ruins para a fila indicada pelo conector “Nao”.

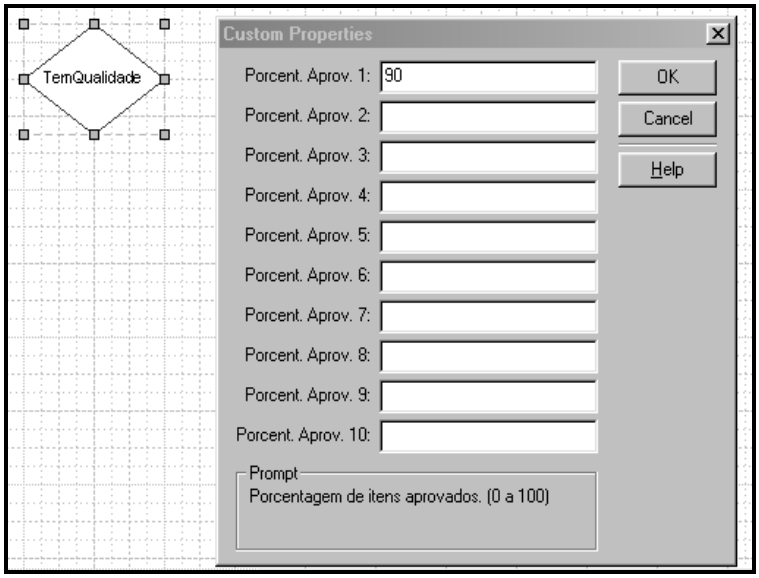


Figura 5.18 Bloco Inspeção "TemQualidade" utilizado no exemplo de uma máquina que fabrica peças, onde se indica a porcentagem de peças Aprovadas na Inspeção; os outros 10% são reprovadas e devem ser reprocessadas.

Propriedade:

Porcent. Aprov.: um número de 0 a 100 que representa a porcentagem de entidades que serão aprovadas, sendo encaminhadas para uma fila diferente do que as peças reprovadas. A fila para onde as peças aceitas vão deve-ser indicadar por um conector to tipo Sim: Sim1, Sim2, Sim3 etc.

Observações:

(1) O conector "Nao" está associado automaticamente à faixa de itens que não são aprovados nas faixas de inspeção. Assim sendo, caso haja duas faixas de inspeção, uma com porcentagem 20% e outra com 25%, a probabilidade de reprovação será de 55%. A primeira faixa está relacionada com o conector "Sim1", a segunda com o "Sim2" e os itens reprovados com o "Nao".

(2) Para a primeira faixa pode-se usar tanto o conector "Sim" quanto o conector "Sim1", pois ambos são aceitos pelo SimVisio.

(3) Para criar um conector "Sim(N)", com N variando de 1 a 10, basta clicar sobre um conector qualquer e digitar o texto, como por exemplo "Sim7".

8º BLOCO: AUTO

Este bloco deve ser colocado logo após o término de uma atividade e realiza a busca automática da fila de menor tamanho dentre aquelas que estão disponíveis.

Exemplo: Em um modelo que simula um pequeno supermercado, os clientes chegam em intervalos exponenciais de média 80 s. Os clientes pegam a mercadoria em um carrinho, atividade com duração média de 1000 s e desvio padrão de 300 s segundo uma distribuição Normal. Depois ele escolhe a menor fila dentre as quatro correspondentes às caixas do mercado. Cada caixa gasta em média 300 s com desvio padrão de 50 s segundo uma distribuição normal.

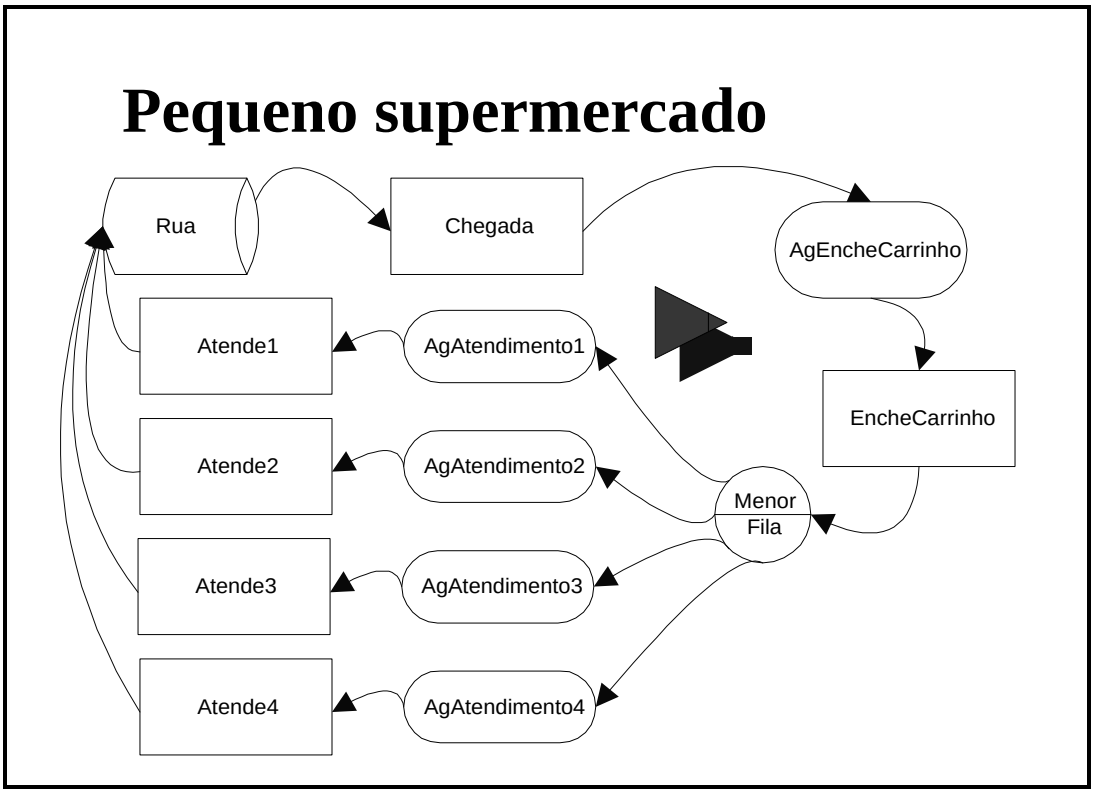


Figura 5.19 Exemplo de um pequeno supermercado, onde o bloco Auto "MenorFila" escolhe automaticamente a menor fila no momento ao final da atividade EncherCarrinho.

5.10 Alguns artifícios para construção de modelos em DCA

5.10.1 Filas paralelas

É comum existir situações em que duas ou mais entidades caminham juntas num sistema durante parte do processo simulado, com pelo menos duas atividades comuns consecutivas. Este é o caso, no problema do bar, das entidades cliente e copo que participam juntas das atividades servir e beber.

Nestes casos, é oportuno recordar que as filas são exclusivas de cada classe de entidade. Assim sendo, quando diferentes entidades participam juntas de atividades consecutivas, deve-se ter uma fila intermediária para cada tipo de entidade, ainda que, na prática, elas permaneçam fisicamente juntas. O sincronismo entre as entidades é, no caso, assegurado pela própria lógica do modelo.

No caso do bar, ao término da atividade "SERVIR", o cliente passa para a fila "PRONTO" enquanto que o copo passa para a fila "CHEIO", juntando-se novamente na atividade "BEBER". Cabe observar que ambas as filas são do tipo "dummy", servindo apenas como intermediação entre duas atividades que, na prática, são executadas sem qualquer espera entre elas.

Filas com esta característica de espera (a mesma para várias entidades) são denominadas filas paralelas.

5.10.2 Priorização de atividades e entidades

Há casos em que a ordem de teste para o início das atividades é relevante. Isto ocorre quando duas ou mais atividades competem por um mesmo recurso, recurso este representado por uma entidade. Nestes casos, é necessário definir uma prioridade relativa entre as atividades. Isto é feito indicando-se o valor da prioridade na propriedade correpondente do bloco Atividade. Ele caracteriza a ordem de teste das atividades, usando-se o valor 1 para a de maior prioridade.

No caso do bar, as atividades "Servir" e "Lavar" competem entre si pelo recurso "Garçom". Para dar maior prioridade à atividade "Servir" (veja figura 5.11), ela tem um número de prioridade menor (1) que a atividade "Lavar" (2). Observe ainda que, apesar de definida, a prioridade das demais atividades não é relevante.

Quando a ordem de teste do início das atividades não é relevante, como no caso do problema das sondas, não há necessidade de se incluir este detalhe num DCA.

Quando for conveniente ou necessária a atribuição de prioridades diferentes para entidades de uma mesma classe, existem duas opções a seguir:

- Diferenciando no próprio DCA as entidades com diferentes prioridades. Neste caso, o DCA torna-se mais complexo, pois são definidas novas filas e atividades para cada entidade criada.
- Não diferenciando as entidades no DCA. Neste caso, as entidades são priorizadas em função de um ou mais atributos. Esta abordagem resulta num DCA mais simples, mas em uma maior complexidade no processamento da simulação pelo computador.

5.10.3 Artifício para controle de horário ou bloqueio de recurso

Quando se deseja bloquear temporariamente um recurso ou estabelecer um controle de horário, o artifício a ser usado consiste na definição de uma nova entidade lógica, denominada genericamente de "Horário".

O ciclo de vida desta nova entidade compõe-se de duas atividades: uma correspondente ao período em que o recurso controlado está disponível e outra relativa ao período em que este recurso está bloqueado. Consequentemente, duas filas (uma delas "dummy") serão criadas.

O elo de ligação entre esta entidade lógica de controle e o ciclo de vida da entidade que corresponde ao recurso controlado, consiste na atividade de bloqueio. Um exemplo ilustra o uso deste artifício:

Suponha que, por razões trabalhistas, o garçom do bar tenha direito a um intervalo para descanso. Desta forma, "Descansar" seria uma nova atividade do garçom. O início da atividade "Descansar" ocorre assim que o garçom ficar ocioso pela primeira vez, após chegar a hora do descanso. Introduzindo a entidade lógica "Horário", como mostra a figura 5.16, esta situação fica perfeitamente representada.

De fato, ao início da simulação a entidade Horario (quantidade = 1) também inicia a atividade "Trabalhar" cuja duração corresponde a um turno de trabalho. Enquanto isso, o garçom divide sua atividade entre servir os clientes, lavar os copos ou mesmo ficar ocioso. Ao término da atividade "Trabalhar", quando o garçom tem direito a um intervalo, a entidade Horario passa para a fila "PreDescansar". Dando maior

prioridade à atividade "Descansar" que às demais, assim que o garçom voltar da fila "Ocioso" pela primeira vez, ele iniciará esta atividade. Ao término do descanso, o garçom volta à fila "Ocioso" para repetir sua rotina de trabalho, enquanto que a entidade Horario passa para a fila "PosDescansar". Esta última fila é "dummy". Assim sendo, inicia-se imediatamente um novo ciclo de trabalho, com uma nova execução da atividade "Trabalhar". Desta forma, consegue-se bloquear temporariamente recursos numa simulação.

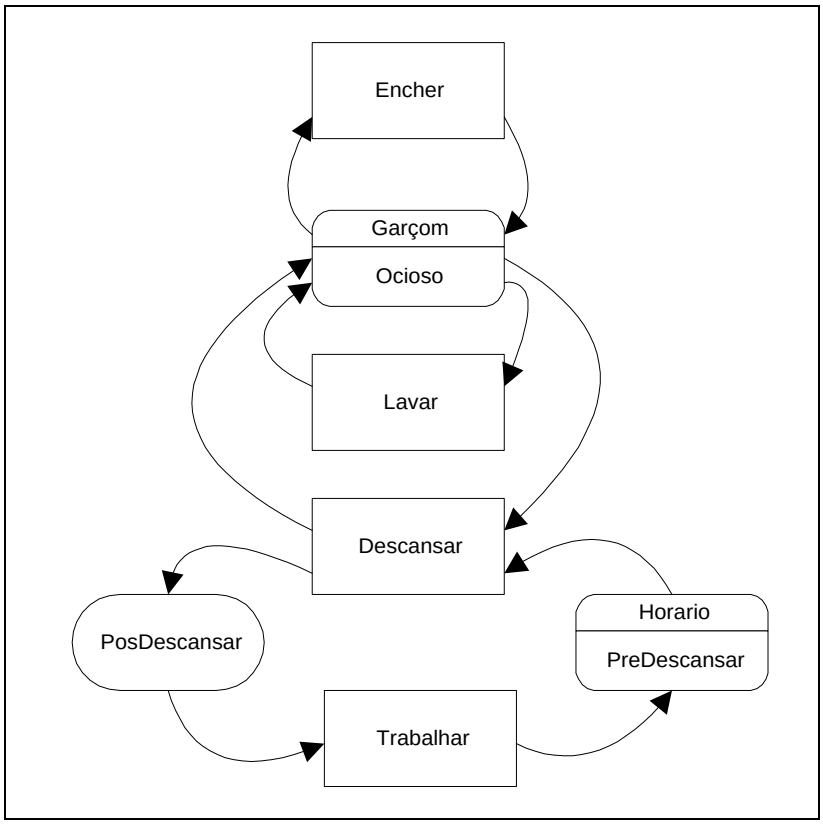


Figura 5.20 Artifício para controle do horário do garçom

5.10.4 Disciplina das filas

Por convenção, a disciplina FIFO ("First In, First Out") é considerada padrão ("default") numa simulação. No entanto, outras disciplinas poderão ser especificadas, como a LIFO ("Last In, First Out"), Por Prioridade, Maior Atributo ou Menor Atributo. Nos dois últimos casos deve-se indicar o nome do Atributo a ser avalidado.

5.10.5 Condição de início de uma atividade

Para finalizar é interessante ressaltar que o início de uma atividade só acontece quando são satisfeitas todas as condições iniciais pré-estabelecidas. No caso do bar, por exemplo, a atividade "Encher" só pode ser iniciada caso as seguintes condições sejam satisfeitas:

- Pelo menos um cliente na fila "AgEncher";
- Pelo menos um garçom na fila "Ocioso";
- Pelo menos um copo na fila "Limpo".

Por outro lado, a atividade "Beber" requer um cliente na fila "AgBeber" e um copo na fila "Cheio" e mais nada. Assim, tão logo termine a atividade "Servir", a atividade "Beber" se inicia, pois ambas as condições serão imediatamente satisfeitas.

5.11 Erros mais comuns na confecção de DCA's

Embora seja aparentemente simples a construção de um DCA, a prática mostra que muitos erros são cometidos na sua elaboração, particularmente por usuários menos experientes com esta abordagem de modelagem. Os erros mais comuns são:

- Misturar entidades de diferentes tipos numa mesma fila;
- Inexistência de fila entre duas atividades consecutivas;
- Ciclo de vida aberto;
- Desvios condicionais saindo de filas.

É interessante informar que por questões de clareza, desaconselha-se que duas entidades percorram um bloco de atividade em sentidos opostos.

5.12 Limitações do DCA

Antes de concluir o estudo do DCA, cabe mencionar que, apesar da sua utilidade como linguagem simbólica para a descrição do comportamento dinâmico de um sistema, um DCA nem sempre é capaz de descrevê-lo por completo. Nestes casos, espera-se que o DCA proporcione uma descrição aproximada do sistema, servindo

como um ponto de partida para o seu estudo. Assim, ficará a cargo do usuário incluir no seu modelo os detalhes do sistema não captados pelo DCA.

Dentre as principais limitações atuais do DCA, pode-se citar:

- Impossibilidade de representar a suspensão de atividades em andamento;
- Impossibilidade de representar atividades com durações condicionadas a outros eventos;
- Impossibilidade de reprogramar uma atividade em andamento;
- Impossibilidade de representar regras de decisão mais complexas para o início de atividades.

5.13 Considerações sobre a criação de novos modelos.

A construção de um DCA representa a etapa mais desafiante e criativa de todo o processo de modelagem para a simulação a eventos discretos. Ela deve ser, portanto, devidamente valorizada pelo usuário. Embora aparentemente simples, a construção de um DCA requer muitas vezes uma boa dose de engenhosidade. Assim sendo, o melhor conselho a se dar ao usuário é que ele desenvolva ao máximo esta habilidade, resolvendo o maior número possível de exercícios de modelagem.

Capítulo 6
Exemplos de aplicação do SimVisio

Mostramos a seguir alguns modelos desenvolvidos para a validação do sistema SimVisio. Os resultados foram analisados e comparados com outros, obtidos de programas de simulação tradicionais como o Arena e o Simul.

Exemplo 1: O problema do Teatro.

Durante o dia o bilheteiro de um teatro se encarrega da venda de ingressos e do atendimento a consultas por parte do público. As consultas pessoais têm prioridade sobre as chamadas telefônicas que, graças a um sofisticado sistema eletrônico, são colocadas numa fila e atendidas na ordem de chegada. O público é educado e muito paciente, nunca indo embora ou desligando o telefone sem que tenha sido atendido. O intervalo entre a chegada de duas consultas pessoais consecutivas segue uma distribuição exponencial com média de 12 unidades de tempo. O intervalo entre a chegada de duas chamadas telefônicas consecutivas também é exponencialmente distribuído com média 10 unidades de tempo. Um atendimento pessoal leva em média 6 unidades de tempo, seguindo uma distribuição exponencial. Uma chamada telefônica leva em média 5 unidades de tempo, distribuindo-se também segundo uma exponencial. Deseja-se simular este problema para estimar a taxa de ocupação do atendente e o tempo de espera das pessoas.

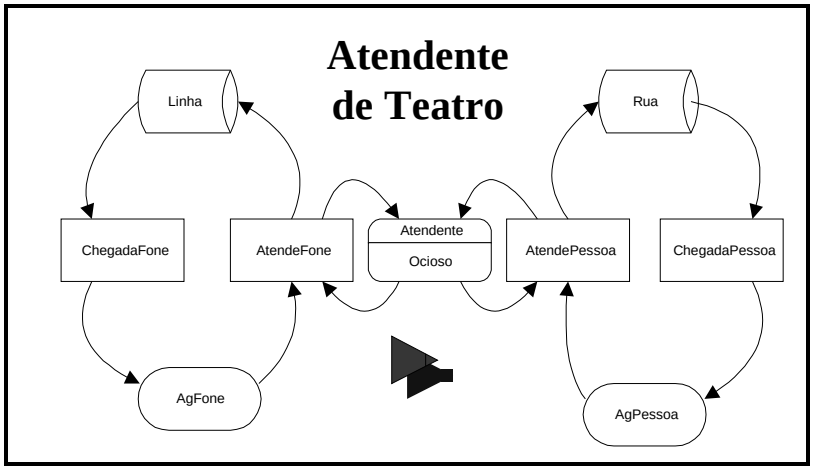
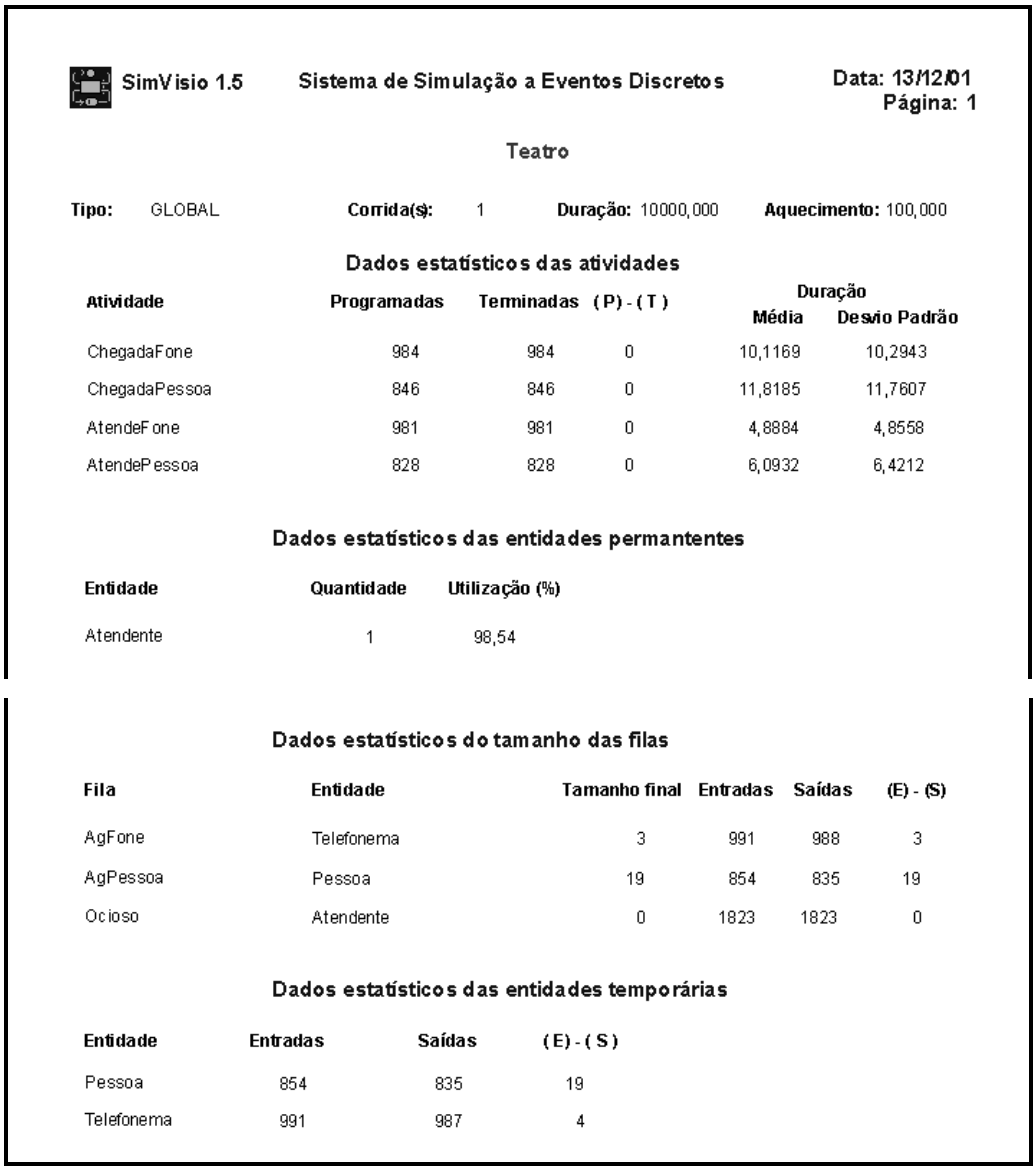


Figura 6.1 Modelo para o problema do atendente de teatro.

Abaixo mostramos o relatório gerado pelo SimVisio para o problema do teatro.



SimVisio 1.5 Sistema de Simulação a Eventos Discretos Data: 13/12/01 Página: 1

Teatro

Tipo: GLOBAL **Corrida(s):** 1 **Duração:** 10000,000 **Aquecimento:** 100,000

Dados estatísticos das atividades

Atividade	Programadas	Terminadas	(P) - (T)	Duração Média	Duração Desvio Padrão
ChegadaFone	984	984	0	10,1169	10,2943
ChegadaPessoa	846	846	0	11,8185	11,7607
AtendeFone	981	981	0	4,8884	4,8558
AtendePessoa	828	828	0	6,0932	6,4212

Dados estatísticos das entidades permantentes

Entidade	Quantidade	Utilização (%)
Atendente	1	98,54

Dados estatísticos do tamanho das filas

Fila	Entidade	Tamanho final	Entradas	Saídas	(E) - (S)
AgFone	Telefonema	3	991	988	3
AgPessoa	Pessoa	19	854	835	19
Ocioso	Atendente	0	1823	1823	0

Dados estatísticos das entidades temporárias

Entidade	Entradas	Saídas	(E) - (S)
Pessoa	854	835	19
Telefonema	991	987	4

Figura 6.2 Relatório para o problema do atendente de teatro.

Através dos dados do relatório podemos tirar algumas conclusões interessantes. A taxa de utilização do atendente é de quase 100%. De fato, o número de pessoas aguardando atendimento ao final da simulação é bastante alto: 19 pessoas. Para problemas de tamanho de corrida maior verificamos que o tamanho da fila "AguardaAtendimento" tende a aumentar ainda mais, o que mostra que o sistema está em desequilíbrio.

Exemplo 2: O problema do "Flow shop".

O exemplo estudado teve por base o artigo de Barnett e Barman (1986). Trata-se de um "flow shop" simplificado constituído por dois centros de serviços, cada centro com duas máquinas idênticas. A figura 6.3a mostra o fluxo de peças pelos centros de serviços.

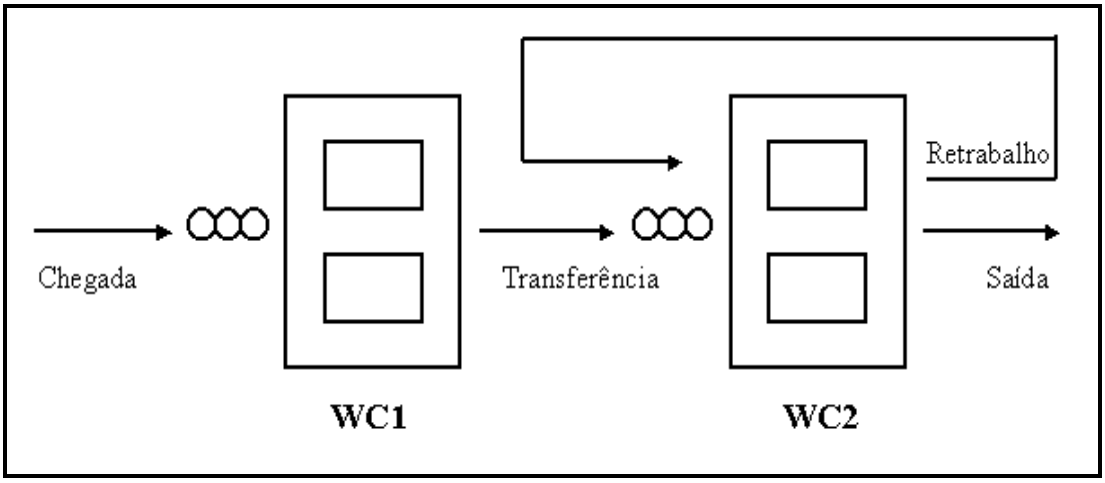


Figura 6.3 Esquema do Flow Shop Simulado

Individualmente, os serviços chegam ao primeiro centro de serviço (WC1) a intervalos aleatórios, formando uma fila de espera caso encontrem suas máquinas ocupadas. Com base na regra de atendimento utilizada para este centro, os serviços são selecionados para processamento, tão logo uma máquina fique disponível. Terminado o processamento em WC1, cuja duração é aleatória, um serviço passa para o centro 2 (WC2); esta transferência de um centro para outro leva um tempo constante de 0.30 unidades de tempo. Sabe-se ainda que cerca de 10% dos serviços completados (saindo de WC2 pela primeira vez) devem ser parcialmente refeitos, sendo reprocessados uma única vez em WC2 com um tempo de processamento adicional constante de 0.10 unidades de tempo; os serviços a serem refeitos têm maior prioridade que os demais já aguardando na fila, sendo sempre atendidos em primeiro lugar.

Supõe-se que o tempo de processamento de um serviço em cada centro seja conhecido quando da sua chegada ao sistema. Os tempos de processamento em cada centro são independentes. O intervalo entre chegadas consecutivas de serviços segue uma distribuição exponencial com média 1.068 unidades de tempo. Os tempos de atendimento em WC1 e em WC2, que são independentes, seguem uma distribuição normal com média de 1.84 unidades de tempo, a mesma para ambos os centros. Os

valores das médias dos tempos de atendimento e do intervalo entre chegadas foram escolhidos de modo a resultar numa ocupação esperada de cada centro da ordem de 87%. Foram considerados três diferentes valores para o desvio padrão dos tempos de atendimento: 0.3, 0.6 e 0.9 unidades de tempo, correspondentes a um nível de variabilidade baixo, médio e elevado. A data devida de cada serviço (Due Date) foi definida com base na fórmula:

Due Date = Data de Chegada + (1+k) x (tempo total de atendimento),

Nesta fórmula o valor adotado para k foi 1, enquanto que o tempo total de atendimento correspondeu à soma dos tempos efetivos de processamento do serviço em cada centro, sem levar em conta eventuais reprocessamentos.

Cada corrida de simulação teve a duração de 10.000 unidades de tempo, correspondente a cerca de 9500 serviços sendo processados, com um período prévio de aquecimento de 100 unidades de tempo (não incluído na duração da corrida).

O diagrama do ciclo de atividades para este "flow-shop", que descreve graficamente a dinâmica do sistema, é apresentado na figura 6.3.

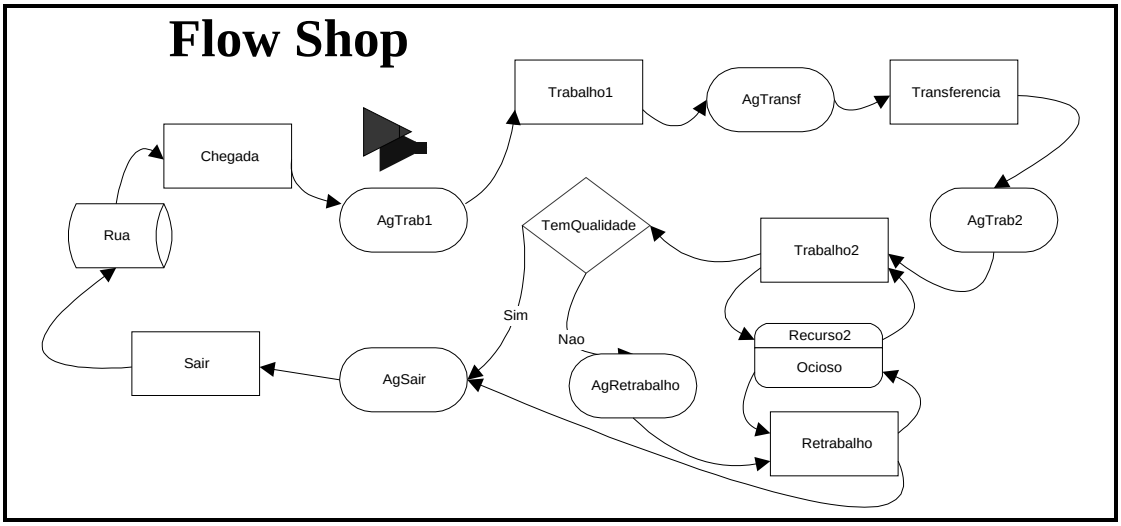


Figura 6.4 Modelo para o problema do Flow Shop.

Para este problema criamos alguns atributos associados à entidade “Cliente”, criada pela fonte “Rua”; estes atributos são: Time1, Time2, Lateness, Tardiness, DueDate e Ttotal. Para defini-los basta colocar estes mesmos nomes na propriedade “Atributo” do bloco “Rua”, tomando o cuidado de separar os atributos por “&” ao invés de utilizar a vírgula “,”. Os atributos Time1 e Time 2 estão associados à duração das

atividades “Trabalho1” e “Trabalho2”. No nosso caso optamos por não indicar a duração diretamente na propriedade “Distribuição” do bloco “Atividade” pois assim podemos realizar melhor as operações matemáticas necessárias para o cálculo dos atrasos que o problema pede.

Exemplo 3: O problema do supermercado

O sistema de atendimento no supermercado é de múltiplos servidores: os caixas estão localizados lado a lado e funcionam sob a regra PEPS (primeiro a entrar, primeiro a sair). O processo do atendimento é constituído de três etapas: registro de mercadorias através da leitura dos códigos de barras, pagamento e empacotamento. Quando um cliente chega a um caixa com seu carrinho de compras, ele precisa optar por um dos checkouts para ser atendido. Esta escolha pode ser feita pelo critério do tamanho da fila, da localização e/ou do operador que realizará o atendimento. De qualquer forma, tendo escolhido um dos checkouts, o cliente entra na fila, caso haja, e, quando chega a sua vez, coloca as mercadorias na esteira. É neste momento que o cliente percebe o início do atendimento, pois começa a desempenhar o seu papel no processo. Os consultores observaram as atividades dos funcionários, e constataram que elas dependiam da forma de pagamento. O cliente pode pagar com dinheiro, cheque, cartão de crédito, cartão de banco (débito automático em conta corrente) ou Cupom (Ticket alimentação). Neste último caso, a complementação do valor da compra pode ser feita por qualquer outra forma de pagamento.

Construa um modelo de simulação para este problema. O estudo tem por objetivo avaliar a qualidade do serviço oferecido (tempo de espera na fila) para diferentes taxas de utilização e para duas alternativas de configuração do sistema de atendimento (fila única x filas múltiplas). Em todos os casos consideraremos uma bateria de quatro caixas, todas com desempenho similar. A duração da simulação deverá ser de, pelo menos, 144000 segundos (40 horas).

a) Fila única sem diferenciar os clientes:

Sem diferenciar os clientes segundo a forma de pagamento, considere um sistema de fila única de atendimento. Neste caso, a duração da atividade de atendimento será a soma dos tempos das três etapas do processo: registro, pagamento e

liberação do cliente. Supondo independência, considerar as seguintes distribuições de probabilidade para estes tempos em segundos: Registro- LogNormal (156, 333), Pagamento - LogNormal (103, 88) e Liberação - LogNormal (56, 100). Fazer duas corridas: uma para um movimento médio de clientes (intervalo entre chegadas exponencial com média de 100 segundos); outra para um movimento forte de clientes (intervalo entre chegadas exponencial com média de 80 segundos).

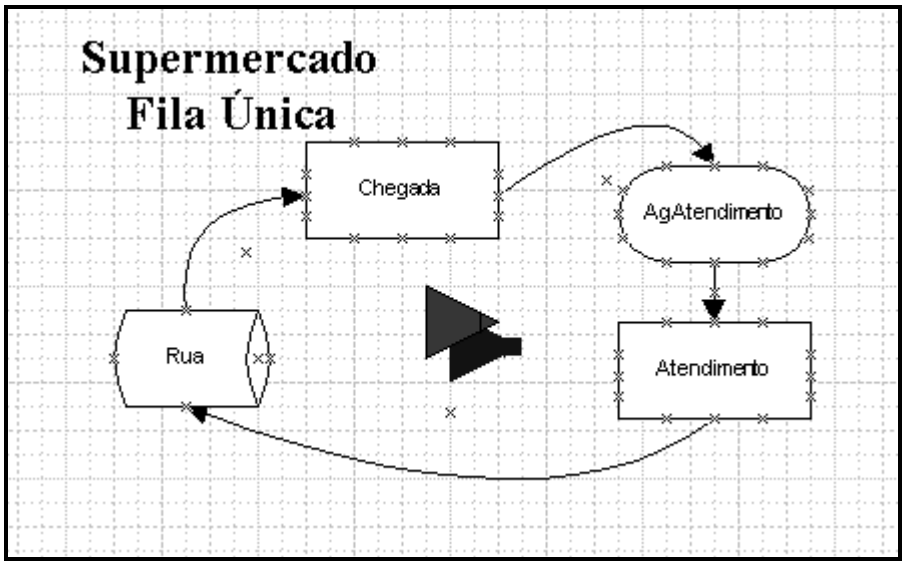


Figura 6.5 Modelo para o problema do Supermercado para fila única.

b) Fila única diferenciando os clientes:

Diferencie agora os clientes em função da forma de pagamento e, conseqüentemente, o respectivo tempo de pagamento (Tpag). Mantenha, porém, as mesmas distribuições para o tempo de registro e para o tempo de liberação. Considere ainda que 53% dos clientes pagam em dinheiro [Tpag ~ LogNormal (55, 31)], 33% através de cartão de crédito ou débito automático [Tpag ~ LogNormal (133, 52)], e 14% através de cheque ou ticket alimentação [Tpag ~ Erlang (107, 2), (Média = 214)]. Refaça a simulação para um movimento médio de clientes (intervalo entre chegadas exponencial com média de 100 segundos).

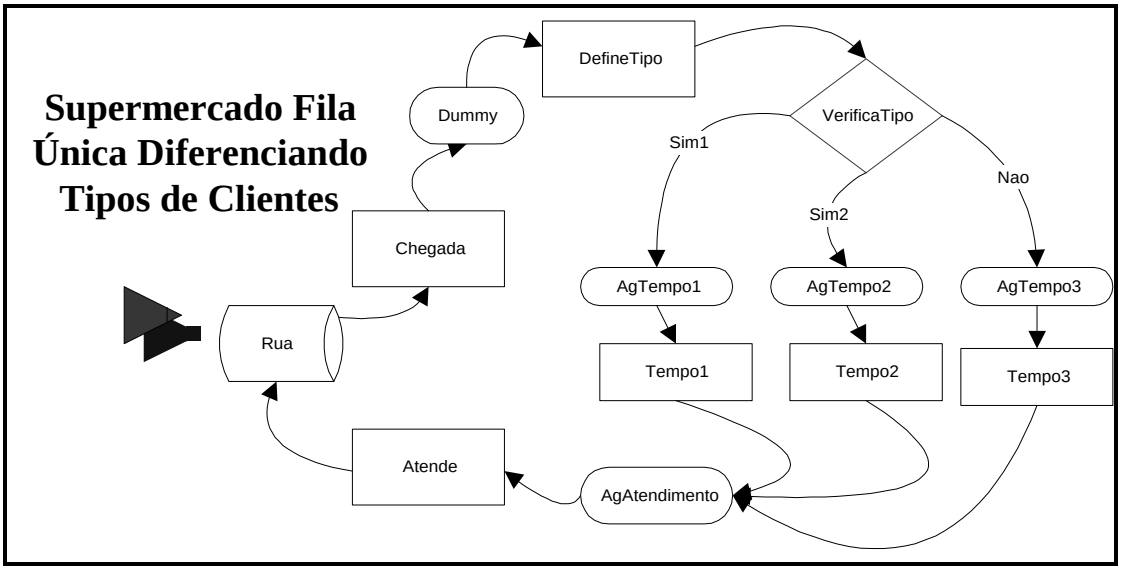


Figura 6.6 Modelo para o problema do Supermercado para fila única com diferenciação de clientes por tipo de pagamento.

c) Fila múltipla diferenciando os clientes:

Refaça o item anterior, considerando agora um sistema de filas múltiplas de atendimento no qual o cliente tem como critério de escolha do servidor, aquele que estiver livre ou com a menor fila no momento de sua chegada à estação de atendimento. Após a escolha de uma fila, não será mais permitida a troca de filas, ainda que um servidor seja liberado.

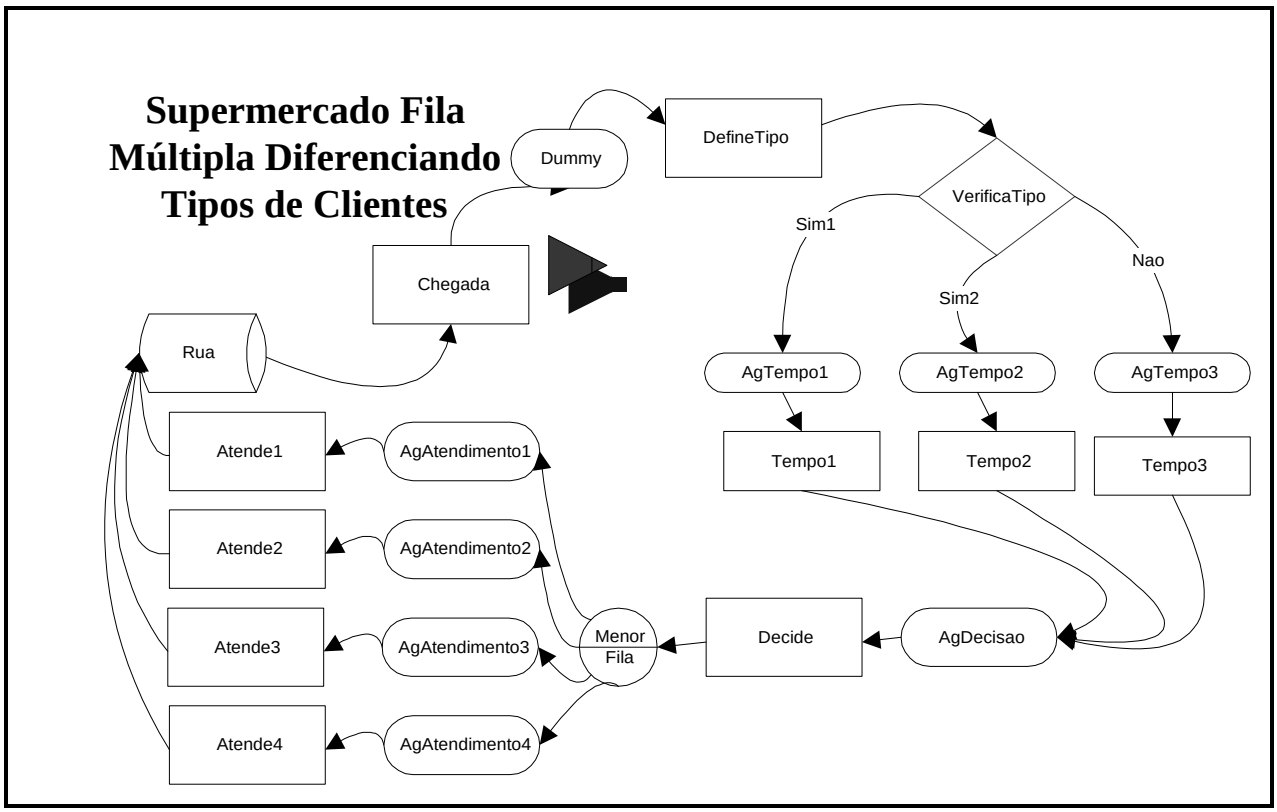


Figura 6.7 Modelo para o problema do Supermercado para fila múltipla com diferenciação de clientes por tipo de pagamento e escolha da fila de menor tamanho.

Capítulo 7
A programação manual no SimVisio

O SimVisio possui um módulo programável que permite a adaptação do programa à peculiaridades de problemas em que não é possíveis modelar através dos blocos lógicos do DCA.

A programação manual é feita através do compilador Delphi, gerando um programa que interage com o modelo lógico criado no ambiente SimVisio. Para possibilitar a identificação dos ciclos de atividades, deve-se utilizar o bloco “Prog” na contrução do modelo lógico. Este bloco tem por fim indicar ao programa quais as possíveis alternativas de filas que podem se seguir à uma atividade e se comporta durante o período de identificação dos ciclos do mesmo modo que o bloco “Auto”.

Para exemplificar o recurso de programação manual vamos utilizar o exemplo do Call Center, cujo enunciado mostramos a seguir.

Um Call Center recebe ligações telefônicas a intervalos entre chamadas que seguem uma distribuição exponencial com média de 60 segundos (supor constante ao longo do período). As chamadas têm duração independente, segundo uma distribuição Erlang com parâmetros k=3 e m=90 segundos (média da distribuição = 270 segundos). O Call Center dispõe de Na=6 atendentes continuamente em serviço e uma capacidade de espera de Esp=5 chamadas (ouvindo aquela musiquinha chata!). Simule este problema e avalie o seu desempenho e padrão qualidade do serviço (ocupação, tempo de espera, tamanho da fila de espera, percentual de chamadas perdidas, tempo total de atendimento). Duração mínima da corrida: 300000 segundos. Teste diferentes configurações do sistema variando o número de atendentes (Na) e a capacidade de espera (Esp), avalie o desempenho do sistema e sugira o que seria, a seu ver, a melhor configuração de operação para os dados fornecidos. Sugestão: Na = 5 a 7, Esp = 4 a 6. A peculiaridade deste problema é que, quando é atingido o limite de 5 chamadas que esperam na fila ouvindo a música, as novas chamadas são desviadas para um sistema de reprodução de uma mensagem que diz: “No momento os nossos atendentes estão todos ocupados. Tente mais tarde”.

O trabalho de programação manual deste problema envolve duas etapas. Em primeiro lugar deve-se indicar no diagrama lógico as possíveis filas que podem se

seguir à atividade “Entrada”. Utilizamos para este fim o bloco “Prog1”, como é mostrado a seguir.

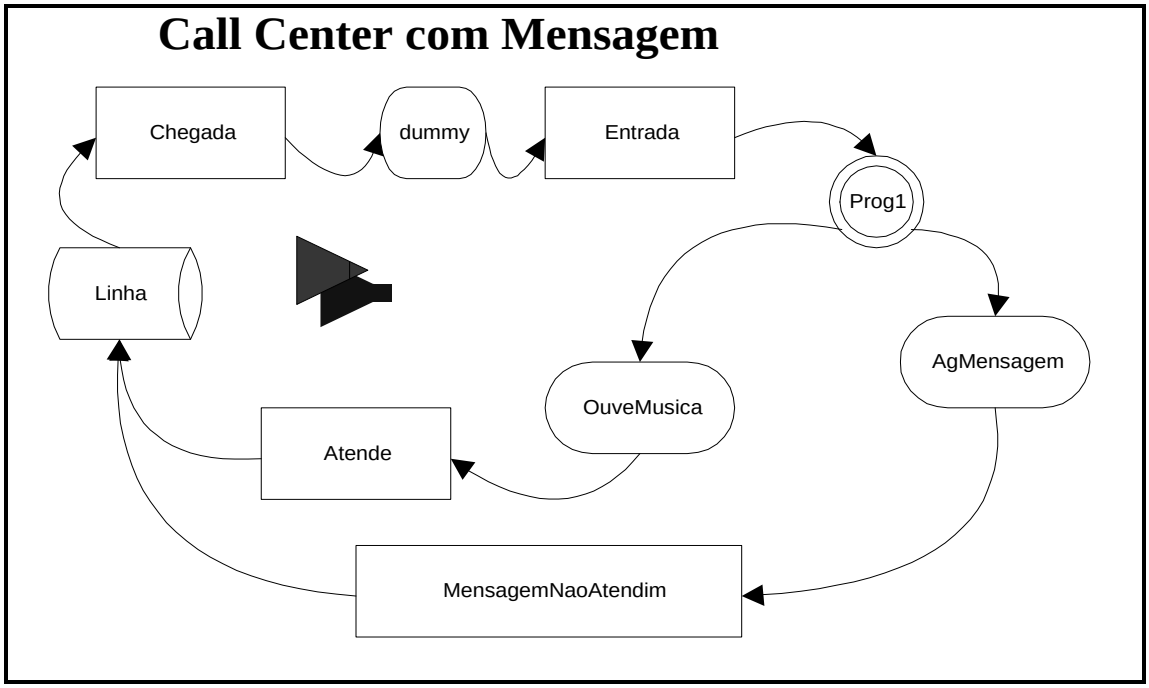


Figura 7.1 Modelo para o problema do Call Center.

A segunda etapa envolve o código do programa SimVisio.exe que precisa ser modificado no ambiente do Delphi. Na prática isto é feito abrindo-se o Projeto “SimVisio.dpr” no compilador Delphi e editando o código no módulo “ProgManual.pas”, que é uma das unidades do nosso programa. Como o programa deve ser compilado com alterações no seu código para este problema específico, deve-se ter o cuidado de guardar os códigos originais do programa em um diretório reservado e somente trabalhar em cópias. Deste modo, outras alterações de código para problemas diferentes poderão ser feitas com base no diretório original.

Voltemos ao nosso exemplo. Para o problema do Call Center é necessária a programação manual do caminho que a chamada deve percorrer após a atividade “Entrada”; se a fila “OuveMusica” tiver mais de 5 elementos, ela deve ser direcionada para a outra fila, cujo nome é “AgMensagem”, encaminhando-se depois para a Mensagem de não atendimento (atividade “MensagemNaoAtendim”).

Sabemos que o fim da atividade corresponde a um evento do tipo B; no nosso caso estamos preocupados com o evento B relativo ao fim da atividade de Entrada.

No Delphi devemos programar o código abrir a unidade ProgManual, que define o formulário FProgManual. Isto é feito escolhendo-se o menu View/Units ou teclando Ctrl+F12 e escolhendo-se “ProgManual”.

Nesta unidade existem dois procedimentos previamente declarados que são:

- Procedure Inicio_Atividades_Condicionais;
- Procedure Fim_Atividades_Condicionais (Num_B : Integer).

O primeiro deve ser usado quando queremos programar manualmente o início de uma atividade (Evento C) enquanto que o segundo serve para programar o fim das atividades (Evento B), e é justamente este que será utilizado por nós.

A programação dos procedimentos acima citada segue regras preestabelecidas, e utilizam uma série de variáveis globais, funções e rotinas definidas na biblioteca do SimVisio.

A programação do término da atividade deve acontecer distribuindo-se as entidades pelas filas subseqüentes a ela. Abaixo está a programação do fim da atividade Entrada do nosso exemplo.

```
Procedure Fim_Entrada;
var Fila1,Fila2:TFila;
Begin
  Fila1:=FModelo.FindComponent('OuveMusica') as TFila;
  Fila2:=FModelo.FindComponent('AgMensagem') as TFila;
  If TamanhoFila(Fila1)>5
  then
    ColocaNaFila(EntCorrente,Atras,Fila2)
  else
    ColocaNaFila(EntCorrente,Atras,Fila1);
end;
```

Este procedimento deve então, ser chamado a partir de Fim_Atividades_Condicionais.

```
Procedure Fim_Atividades_Condicionais(Num_B : Integer);
var FilaCor : TFila;
    NomeATv, NomeEnt : String;
begin
```

```
Atv_Ent_Fila_Evento_B(Num_B,NomeAtv,NomeEnt,FilaCor);
If (UpperCase(NomeAtv)='ENTRADA') and UpperCase(NomeEnt)='CHAMADA')
    then Fim_Entrada;
end;
```

Utilizamos o procedimento Atv_Ent_Fila_Evento_B(NumEvento : Integer; Var NomeAtv, NomeEnt : String; Fil : TFila) que informa, a partir do número do evento, o nome da atividade que está a terminar (NomeAtv), o nome da entidade que está sendo liberada (NomeEnt) e a sua fila de destino (FilaCor). Note que é feito um teste para cada evento B para verificar se se trata do evento correto, ou seja, o correspondente à atividade “Entrada” e à entidade “Chamada”. É importante notar que caso haja vários Eventos B de término programado manualmente será necessário programar várias linhas do mesmo padrão, uma para cada evento B, distinguindo-se também os eventos correspondentes às diferentes entidades que percorre a atividade, caso haja mais de uma.

Capítulo 8
A construção do programa SimVisio

8.1 O modo como foi construído

O SimVisio foi construído sobre a base do Simin (Pinto, 1999), ao qual foram feitas várias adaptações, sendo também acrescentados alguns módulos. Os dois principais módulos acrescentados foram o de importação dos shapes do Visio e o de verificação de sintaxe do modelo DCA.

O sistema que foi desenvolvido utiliza o Visio como inteface de entrada de dados. O programa que lê as informações da interface e roda a simulação foi desenvolvido no Delphi e se chama SimVisio.exe. Este programa é composto por vários módulos, que são basicamente formulários (.DFM) e units em Pascal (.PAS). As principais units do SimVisio são:

Tabela 7.1 Principais Units do programa fonte do SimVisio

Unit (Delphi)	Função
1. Util.pas	Principais rotinas do processamento da simulação: coordena o Método da Três Fases.
2. Rotinas.pas	Rotinas para tratamento de filas, atributos e eventos
3. Gerenciador.pas	Implementação das Fases B e C do Método das Três Fases
4. Amostra.pas	Rotinas de amostragem aleatória simples.
5. Estatistica.pas	Monta estatísticas das filas, entidades e atividades para histogramas e relatórios
6. TelaRelGlobais.pas	Tela de criação de relatórios contendo as estatísticas da simulação
7. TelaGrafHist.pas	Tela de criação de histogramas solicitados pelo usuário.
8. Compnt.pas	Componentes de Simulação: Filas, Atividades, Fontes, Entidades, Condição, Inspeção, e Auto.

9. TelaPrincipal.pas	Tela principal do programa, contendo o módulo de leitura do modelo DCA, verificação de erros de modelagem, criação dos componentes de simulação e chamada das rotinas de simulação.

Descrevemos as oito principais *units* do programa. Há outras que não foram descritas, pois são de caráter secundário e auxiliar.

As sete primeiras *units* citadas são praticamente as mesmas do Simin (Pinto, 1999). O Simin, por sua vez, aproveitou muito do que foi desenvolvido no Simul, tendo várias alterações para que o código em Turbo Pascal do Simul pudesse ser aproveitado no Delphi, que é a linguagem do Simin.

A oitava *unit*, Compnt.pas, possui os componentes de simulação. Nela estão alguns componentes originais do Simin, mas com algumas funcionalidades a mais que foram desenvolvidas para o novo programa. Foram também acrescentados alguns componentes novos que não existiam no Simin como, por exemplo, os componentes FilaRec, Inspeção, Condição e Auto.

A nona *unit*, TelaPrincipal.pas, está associada ao formulário principal do sistema. Nela estão os menus de solicitação de dados da simulação, mostrados através de relatórios e de histogramas. Sobre o formulário principal original, criado no Simin, foi acrescentado um botão “Carregar Modelo”, que quando é acionado inicia o processo de leitura, interpretação e análise do modelo a partir do Visio. Para que ele funcione é necessário que o modelo DCA esteja aberto e ativado na tela do Visio.

A seguir são descritas em maior detalhe as três etapas de carregamento do modelo. Procuramos seguir a mesma ordem em que são realizadas as operações no respectivo código fonte em Pascal-Delphi.

1ª. etapa: Lendo os shapes do Visio e criando os componentes

a) Lendo dados da simulação.

Em primeiro lugar o programa procura o Shape **Simulacao** do Visio, onde estão as propriedades de controle de simulação. A cada valor de propriedade lido a partir do Shape Simulacao é preenchida uma váriável correspondente no SimVisio.

b) Criando Entidades.

A partir dos dados dos shapes **Fonte** e **FilaRec** criam-se componentes do tipo Entidade no SimVisio. Se uma classe de entidades é criada a partir de uma Fonte, é considerada automaticamente do tipo temporário e se é do FilaRec é considerada entidade permanente (informalmente também chamada de recurso). As entidades criadas são componentes Delphi da classe **TEntidade** e são colocados no formulário FModelo.

c) Criando Fontes.

O programa cria também componentes Fonte a partir de cada shape **Fonte** encontrado no modelo em Visio. As fontes são a origem de novas entidades no sistema durante a simulação. Note que no item b) se define a classe de entidade e o computador, durante a simulação, em tempo de execução, cria entidades reais que são instâncias desta classe com base no componente fonte. As fontes criadas são componentes Delphi da classe **TFonte** e são colocados no formulário FModelo.

d) Criando Atributos.

Através dos shapes **Fonte** e **FilaRec** podem-se criar também compondentes do tipo atributo que estão associados às entidades do sistema, para utilização durante a simulação. O sistema identifica os shapes e cria os atributos no SimVisio associando-os automaticamente às respectivas entidades. Diversos atributos podem ser criados, e o programa identifica os elementos de uma lista de atributos pelos separadores “&”. Os atributos criados são componentes Delphi da classe **TAtributo** e são colocados no formulário FDadosBasicos.

e) Criando Histogramas de atributos.

O programa identifica se são pedidos histogramas para os atributos a serem mostrados no final da simulação. Os dados do histograma são lidos a partir do respectivo campo dos shapes **Fonte** e **FilaRec** e possuem o seguinte formato (NomeDoHistograma:Base:Largura) onde Base e Largura são números reais.

Tecnicamente, os histogramas são componentes Delphi da classe **THistograma** e são colocados no formulário FDadosBasicos.

f) Criando Atividades

As **atividades** são shapes do Visio. Para cada shape é criada uma atividade no formulário FModelo do SimVisio. O nome da atividade no SimVisio é o lido a partir do texto que é colocado no shape do Visio. As atividades são da classe **TAtividade**, com várias propriedades que são lidas a partir dos shapes: capacidade, distribuição de probabilidade que rege a duração da atividade, histogramas relacionados etc.

Obs: O uso do Parser

Um importante detalhe construtivo foi de grande importância no que diz respeito à determinação da duração da atividade: o uso de um *Parser*. No SimVisio é possível fazer uma composição de várias distribuições de probabilidade, ao invés de somente uma, como era feito no Simin. Para implementar esta importante funcionalidade foi utilizado um componente *Parser* que realiza estas operações de composição. Este componente está disponível na internet e, como conta com o código fonte disponível, foi possível adaptá-lo para o uso em simulação. Através dele podemos também utilizar operadores matemáticos como +, -, X, / e funções como sen, cos, tan, max, min, etc. Foi necessário um trabalho de programação para distinguir quais funções o Parser deveria tratar como funções matemáticas e quais na verdade são distribuições de probabilidade sobre as quais é necessário gerar um número aleatório. Uma *procedure* faz esta identificação, denominada AchaDistribParser.

Ao final de uma atividade pode haver uma alteração do valor de um atributo de uma entidade que passa pela atividade. Esta atribuição é indicada no shape Atividade segundo uma sintaxe padrão de linguagens de programação, como por exemplo:

Sede=Sede-1.

O programa identifica quais os atributos foram utilizados e realiza a operação a cada término de atividade. São possíveis três atribuições por atividade. Note que as operações de atribuição de novo valor por atributo são lentas do ponto de vista computacional, o que atrapalha um pouco a performance em termos de velocidade de execução da simulação. Isto ocorre porque os valores dos atributos das entidades são

lidos toda vez que uma tarefa é iniciada, para que estes valores possam ser utilizados na operação de atribuição indicada para o final da atividade.

A propriedade Capacidade do shape Atividade pode conter um valor de zero em diante. Se o valor N de capacidade é maior ou igual a um, isto significa, que a atividade estará limitada a N tarefas por vez. Esta limitação de capacidade é criada no simulador na forma de recursos que são associados à atividade. Cria-se uma FilaRec que possui somente duas ligações: as entidades da FilaRec entram na atividade quando ela inicia e voltam à FilaRec depois de terminada. Assim, se a FilaRec possuir N entidades, somente N atividades poderão ocorrer ao mesmo tempo. O nome da entidade permanente associada a uma atividade nestes casos é dado automaticamente pelo programa, no formato “Rec_” + nome da entidade. P.ex.: Rec_Atendimento.

g) Criando Filas

As filas são criadas de modo semelhante às atividades. Cada fila possui um nome que é identificado pelo texto do shape correspondente no Visio. As características da fila são lidas a partir das propriedades do shape: capacidade, o que fazer quando excede a capacidade, disciplina, prioridade, quantidade inicial, histogramas solicitados e suas respectivas características. Os componentes correspondentes às filas no SimVisio são da classe **TFila**.

Note que internamente as filas podem ser criadas tanto quando o programa identifica um shape do tipo **Fila** como também quanto o shape é do tipo **FilaRec**. Na verdade, a distinção entre Fila e FilaRec é importante para a montagem inicial do modelo, visto que o FilaRec cria entidades permanentes no início da simulação e enche a TFila correspondente com um número determinado de entidades, para a partir daí começar a simulação. Durante a execução da simulação, no entanto, o programa trata tanto um shape quanto outro indistintamente como componentes TFila.

h) Criando Condições

O shape **Condicao** do Visio serve para direcionar as entidades para diferentes filas dependendo do valor de um dos seus atributos. São importantes neste caso as propriedades Atributo, operador e valor. Quando o programa lê o nome do atributo automaticamente verifica se o mesmo foi definido em alguma Fonte ou FilaRec, e caso

haja algum problema ele envia um aviso ao usuário, que deve corrigir o problema. A classe deste componente é **TCondicao**.

i) Criando Inspeções

O shape **Inspecao** possui 10 campos de propriedade para determinar as 10 probabilidades associadas aos destinos que possam ter as entidades que passam por ele. O maior trabalho que o programa deve realizar quando identifica um shape Inspecao é avaliar se os dados são coerentes. É preciso avisar o usuário se caso a soma de todos os campos ultrapasse 100% . O componente correspondente no SimVisio é o **TInspecao**.

j) Criando Auto: nós de busca automática da menor fila disponível.

O shape **Auto** se localiza ao final de uma atividade e orienta a entidade que está saindo para a fila de menor tamanho dentro das disponíveis pela frente. É o único tipo de shape do qual o programa não lê nenhuma propriedade interna, mas somente o seu texto para dar nome ao componente **TAuto**.

Observações:

É importante notar que a correspondência entre shapes do Visio e componentes Delphi não é de um para um. Na verdade, um único shape pode gerar vários componentes, como é o caso das Atividades e das Filas.

Esta primeira etapa da leitura do modelo, embora simples do ponto de vista conceitual resultou em um programa bastante extenso, ocupando 800 linhas de código aproximadamente.

2ª. Etapa: Identificação dos ciclos.

Esta é uma das etapas do programa que envolveu provavelmente mais tempo para o seu desenvolvimento. Ela diz respeito à própria lógica do Diagrama do Ciclo de Atividades e exigiu um estudo aprofundado de como são organizadas internamente as informações de um documento criado em Visio. Felizmente o Visio possui uma boa documentação a este respeito, o que facilitou o nosso trabalho.

A leitura das propriedades do diagrama é feita através de uma *Type Library* ou biblioteca que foi importada para o Delphi a partir do Visio.

A identificação dos ciclos é feita diretamente sobre os componentes em Delphi do modelo e não sobre os Shapes do Visio. Dentro da seqüência do programa, os componentes são criados em uma primeira etapa, citada anteriormente. O trabalho direto sobre os componentes leva vantagem sobre os Shapes do Visio, pois toma menos tempo do ponto de vista computacional.

A seguir são mostradas as etapas necessárias dentro do programa fonte do SimVisio para a identificação de Ciclos.

a) Identificando os conectores.

Em primeiro lugar o programa analisa todos os conectores do modelo. No Visio, eles são objetos do tipo "**Connect**" e um possuem a informação de quais objetos estão sendo ligados pelo mesmo. Com isto, se criaram objetos no Delphi do tipo **TLigacao** com as propriedades Origem, TipoOrigem, Destino e TipoDestino cujos conteúdos correspondem aos shapes conector. É copiado também o texto do conector (p.ex. Sim1, Nao) para a propriedade Texto. Estas informações são a base de toda a análise dos ciclos que e feita a seguir.

b) Criação de nós de início de Ciclo e nós de reinício de Ciclo

Depois das ligações criamos objetos do tipo **Nó**. O conceito de nó está relacionado com aqueles Shapes do diagrama que constituem pontos importantes dos ciclos de entidades. Há nós de início e fim de ciclo, além de nós que constituem uma interrupção no ciclo e que por sua vez constituem também continuações de ciclos. Os nós de início de ciclo são aqueles que servem de entrada para novas entidades no modelo e são os correspondentes ao shapes **Fonte** e **FilaRec**.

Existem nós especiais que servem também para identificar os ciclos e que não são a princípio nós de entrada de entidades no sistema. Eles foram criados a partir de shapes identificados no modelo, durante a primeira etapa de identificação. Eles auxiliam nos ciclos que podem continuar por caminhos alternativos, como é o caso de Shapes do tipo **Condição**, **Inspeção** e **Auto**. Nestes casos são criados tantos nós quantos os caminhos alternativos que saem destes shapes. Assim, por exemplo, se de

um nó de Inspeção saem três conectores, um indicando a fila correspondente ao primeiro tipo de aprovação, o segundo para o segundo tipo e o terceiro para a fila que deve ir no caso de rejeição na inspeção, então devem ser criados três nós no modelo. Estes nós não fazem parte do modelo propriamente dito, servindo como auxiliares nesta etapa de identificação de ciclos. Estes nós são na verdade classes especialmente criadas no Delphi para este fim, e são os tipos **TNoCond, TNoInspec e TNoAuto**.

Existe também um tipo de nó especial relacionado aos shapes do tipo FilaRec. Isto porque de um único shape FilaRec podem se originar também vários ciclos. O exemplo do Bar mostra um exemplo deste tipo de utilização do FilaRec. Para este tipo de Shape também criamos nós especiais, que no Delphi são componentes da classe TNoFilaRec.

c) Iniciando ciclos a partir de Fontes e FilaRec

Toda vez que o programa depara com um componente do tipo Fonte ou FilaRec ele reconhece que está diante do início de um ciclo. Se o componente é do tipo Fonte, sabe que é um ciclo de entidade temporária e de é um FilaRec então sabe que é uma entidade permanente. Após reconhecer algum nó de início o programa vai identificando os conectores e outros componentes que formam parte do ciclo, um após o outro até chegar no final do ciclo. Esta busca é simples quando o ciclo contém apenas fontes, atividades e filas. Neste caso o ciclo começa com a Fonte, passa por uma ligação até a atividade de chegada (automaticamente assinalada deste modo por ser a primeira), a seguir pelo conector que leva à primeira fila, a seguir a própria fila, de novo um conector, a atividade correspondente e assim por diante com conectores, filas e atividades até se fechar o ciclo com um conector que está ligado à fonte original.

A busca é interrompida quando se depara com alguns tipos de componentes especiais: FilaRec, Condição, Inspeção ou Auto. Nestes casos a identificação do ciclo é temporariamente interrompida e somente reiniciará quando, mais para frente, o programa procurar por ciclos que se reiniciam a partir de nós especiais.

A partir desta etapa do programa já se identificam as entidades dos ciclos. Assim sendo, se um ciclo passa por uma Fila, o nome da entidade do ciclo é automaticamente preenchido na propriedade Ent da própria Fila. Também se distinguem os ciclos por cores, tendo cada entidade uma cor de ciclo distinta. Pode haver casos que uma mesma entidade participa de mais de um ciclo, e quando isto acontece, os respectivos ciclos

têm a mesma cor. Isto ocorre com ciclos de entidades permanentes que servem a duas ou mais atividades, como é o caso do Garçom no exemplo do Bar, que serve tanto às atividades Servir quanto a Lavar.

d) Reiniciando ciclos a partir de nós especiais

Como foi dito anteriormente, há nós especiais em que a identificação do ciclo é interrompida quando se depara com um deles. O programa identifica cada um destes nós especiais e conta quantos são. A seguir passa por todos eles, um a um e reinicia a identificação da seqüência de blocos que compõem o ciclo. Assim como no caso anterior, esta identificação é suspensa temporariamente quando se depara com um novo nó especial, podendo terminar quando se fecha um ciclo completo, o que ocorre quando se identifica um componente Fonte ou um componente FilaRec.

Nesta etapa também se identificam as entidades e se diferenciam as cores dos ciclos como na etapa anterior.

e) Processando nós e conectores

Esta é uma etapa importante, pois ao mesmo tempo em que se criam os ciclos, se montam as redes de informações de entrelaçamento entre os diversos componentes de simulação para se poder rodar o modelo. Por exemplo, a determinação das propriedades fila_pre e fila_pos do componente Atividade envolve a identificação de que um conector entre uma fila e uma atividade. O processamento é feito um nó de cada vez.

Como exemplo simples, quando o nó que está sendo processado é uma Fila, guarda-se o seu nome para uso posterior na variável global UltimaFila. Segue-se o processamento do ciclo e quando de identifica a Atividade seguinte, se associa a Fila à Atividade através do preenchimento da propriedade Fila_pre da Atividade com o nome da Fila correspondente, que está indicado pela variável UltimaFila. Este preenchimento é feito de modo automático pelo programa.

Existem muitas outras propriedades que são preenchidas do mesmo modo, sempre utilizando variáveis auxiliares. Citamos algumas delas a seguir.

- Componente Atividade: Filas antecedentes, Filas conseqüentes, Fontes antecedentes, Condições, Inspeção ou Auto que estão no fim da atividade.
- Componente Ligação: Cor

- Componente Fila: Entidade
- Componente FilaRec: Entidade
- Componente Condição: Entidade, Destinos Sim e Não.
- Componente Inspeção: Entidade, Destinos Não e Sim de 1 a 10.
- Componente Auto: Entidade, Destinos 1, 2, 3 até 10.

e) Terminando ciclos

Os ciclos sempre terminam em Fontes ou em FilaRec. Caso aconteça que ao final das etapas anteriores algum conector ainda não tenha sido identificado como fazendo parte de algum ciclo, isto significará que há algum ciclo aberto e uma mensagem de erro será mostrada ao usuário.

3ª. Etapa: Análise da sintaxe do modelo.

Ao mesmo tempo em que são realizadas as duas primeiras etapas acima descritas, são informados ao usuário os eventuais erros que possam ser cometidos, tanto no preenchimento das informações quanto na montagem dos ciclos.

a) Erros de preenchimento

O correto processamento da simulação depende do correto preenchimento das propriedades de cada shape do modelo. Algumas podem ficar em branco, mas outras devem necessariamente ser fornecidas com valores de determinado tipo. Para cada propriedade importante é feita uma verificação do valor fornecido pelo usuário, e mensagens de erro são emitidas pelo programa, indicando o tipo de erro cometido, em que parte do modelo ocorreu e se dá possibilidade ao usuário para que corrija o erro, de modo que depois o programa possa voltar a rodar novamente.

b) Erros de construção dos diagramas

É comum a ocorrência de erros na construção de diagramas. Para evitar que isto aconteça, alguns erros mais comuns são identificados pelo programa.

Para esta análise seguem-se as regras de sintaxe básicas para a montagem de um DCA que são verificadas pelo programa. Note que estas regras são uma adaptação das regras apresentadas no início do capítulo 5, montadas de forma a facilitar o trabalho de depuração do modelo no através do computador.

Regra 1: As filas são de uso exclusivo de uma classe de entidades.
Regra 2: Todos os conectores devem estar ligados nas duas pontas.
Regra 3: Um componente não deve estar conectado a si mesmo.
Regra 4: Dois componentes de mesmo tipo não devem estar ligados diretamente entre si.
Regra 5: Cada atividade deve ser precedida por fila ou fonte ou filarec.
Regra 6: Em uma atividade, o número de conectores que entram e que saem é o mesmo.
Regra 7: Os diversos ciclos devem terminar em uma Fonte ou FilaRec.
Regra 8: Todos conectores devem fazer parte de um ciclo de entidade.

De um ponto de vista das instruções que são executadas pelo computador, podemos detalhar melhor as operações em cada uma das verificações. Note que o programa sempre procura identificar os ciclos, partindo de uma fonte ou de uma filarec e seguindo a sequência alternada de conectores e elementos até chegar no final do ciclo, indo a seguir para outro ciclo até acabar o diagrama completo. A cada elemento verificado como parte do ciclo algumas das verificações (1 a 8) são realizada.

Regra 1: As filas são de uso exclusivo de uma classe de entidades.

Quando o programa encontra uma Fila, se é a primeira vez que ela passa a fazer parte de um ciclo, o nome da entidade do ciclo é preenchido no campo correspondente da fila. Se ocorrer de o programa encontrar novamente esta fila, compara a entidade do ciclo da fila com a entidade do ciclo e se houver discrepância, um aviso é enviado para a tela do computador: “Infração da Regra 1”.

Regra 2: Todos os conectores devem estar ligados nas duas pontas.

No início da verificação dos ciclos a primeira etapa é mapear todos os conectores, identificando a sua origem e o seu destino. Caso haja algum conector que não tenha compnentes de origem ou destino, um aviso é mostrado na tela do computador: “Infração da Regra 2”. Note que muitas vezes os conectores podem parecer conectados na tela do computador, mas não o estarem de fato. Para que um conector esteja bem ligado ao shape correspondente, é preciso verificar se a sua extremidade muda para a cor vermelha quando ele é colocado sobre os pontos de conexão do respectivo shape.

Regra 3: Um componente não deve estar conectado a si mesmo.

Na etapa inicial de verificação dos conectores são identificados as origens e os destinos de cada conector, guardando-se os nomes dos componentes respectivos. Caso o nome da origem e do destino seja o mesmo, isto é sinal de que o componente está conectado a si mesmo, e um aviso é enviado pelo computador: “Infração da regra 3”.

Regra 4: Dois componentes de mesmo tipo não devem estar ligados diretamente entre si.

Na verificação de origem e destino de cada conector são preenchidos, além dos nomes dos componentes, os nomes dos seus tipos. Se estes nomes dos tipos forem os mesmos, como por exemplo, duas filas ou duas atividades, é sinal de que o conector está ligando dois componentes de mesmo tipo, e um aviso é enviado ao usuário: “Infração da regra 4”.

Regra 5: Cada atividade deve ser precedida por fila ou fonte ou filarec.

Para cada conector, no preenchimento do campo do tipo de componente de destino, quando este destino é do tipo atividade, verifica-se o tipo do componente de origem e caso não seja do tipo fila, fonte ou filarec, o ciclo fica incompleto de um ponto de vista lógico e é enviada a mensagem: “Infração da regra 5”.

Regra 6: Em uma atividade, o número de conectores que entram e que saem é o mesmo.

O componente atividade possui dois campos numéricos que são preenchidos durante a etapa de identificação dos ciclos: os campos número de entradas e número de saídas. Assim sendo, se ao percorrermos um ciclo a partir de sua origem e deparamos com uma atividade, acrescentamos um ao número de entradas desta atividade. De modo semelhante, se identificamos um conector que sai desta atividade, somamos um ao número de saídas deste componente. Uma vez criados todos os ciclos, verificamos uma a uma cada uma das atividades e caso o número de entradas seja diferente do número de saídas em alguma delas, é sinal de um erro de construção e um aviso é enviado: “Infração da regra 6”.

Regra 7: Os diversos ciclos devem terminar em uma Fonte ou FilaRec.

Um ciclo pode terminar normalmente quando o programa depara-se com uma fonte ou com uma filarec. No entanto, o usuário pode cometer o erro de fazer um ciclo que não se fecha, terminando em um elemento qualquer diferente dos dois citados. Se isto ocorrer, o programa indicará o erro com uma mensagem: “Infração da regra 7”.

Regra 8: Todos conectores devem fazer parte de um ciclo de entidade.

Quando os diversos ciclos são montados, o programa marca cada conector que foi usado, preenchendo o valor do campo TemCiclo do conector com o valor True. Depois que são identificados todos os ciclos, o programa varre todos conectores e verifica se algum tem o valor de TemCiclo igual a False, o que indica que ele não pertence a nenhum ciclo, e um aviso é mostrado: “Infração da regra 8”.

8.2 Considerações sobre a performance do sistema

Alguns fatores de ordem técnica ocasionaram em limitações de velocidade e de ausência de certos recursos visuais no sistema criado. É muito demorada a etapa de

leitura do modelo, ou seja, a passagem das informações do diagrama da tela do Visio para o programa de simulação, o SimVisio, criado em Delphi que lê as informações do programa Visio. Esta lentidão tem como causa o tipo de tecnologia empregado para a comunicação entre os dois programas que é feita através de uma biblioteca de importação ou "*Type Library*". A baixa performance do processo de importação é inerente à tecnologia de utilizada para esta comunicação (denominada de *OLE Automation),* e somente pode ser resolvida com a utilização de uma plataforma de modelagem mais integrada, de preferência desenvolvida com o mesmo compilador Delphi que foi utilizado para o programa de simulação. Esta última possibilidade foi pensada como uma alternativa para este trabalho, mas não foi levada adiante porque os resultados obtidos com o Visio foram muito bons, salvo a baixa velocidade. Não de descarta a possibilidade de desenvolvimento futuro, após se ter acabado o presente trabalho, o que pode ser feito pelo próprio autor ou por outra pessoa.

Um segundo condicionante de ordem técnica foi a dificuldade de fornecer, na tela do modelo no Visio, algumas informações sobre o andamento da simulação, como contadores para o número de entidades que entram e que saem das filas ou mesmo a animação das entidades percorrendo os ciclos respectivos. Estes recursos, presentes em outros programas de simulação, não foram implementados por problemas de comunicação entre os sistemas utilizados. A comunicação entre o Visio e os programas em Delphi é extremamente lenta por causa da tecnologia de automação Ole. Embora isto permita a leitura de dados para a criação do modelo, que pode ser feito de uma forma lenta, acaba inviabilizando a animação do modelo, quer através mudança de cores de certas partes do diagrama, quer através da escrita de informações na forma de texto na tela do computador. Como o processamento da simulação exige extrema rapidez do programa, esta animação se torna inviável. A criação de uma nova interface de modelagem independente do Visio, conforme foi proposto no parágrafo anterior, poderia resolver este problema também.

De um ponto de vista da capacidade de modelagem, algumas melhorias poderiam ser feitas neste ambiente, principalmente com a introdução de novos blocos no DCA para a representação de agrupamento e desagrupamento de entidades. Estes blocos seriam úteis na modelagem de problemas como o de ambulâncias, onde paciente e equipe de socorro poderiam ser agrupadas e andariam através de uma parte do modelo juntas, separando-se ao final do atendimento. Outro exemplo é o de pessoas entrando

em um elevador em grupo. Atualmente o SimVisio não é capaz de modelar adequadamente estas situações.

Outra função que poderia ser introduzida no sistema no futuro seria a animação gráfica de entidades. Isto poderia ser feito baseando-se na própria tela do modelo, fazendo-se uma representação esquemática em duas dimensões de entradas e saídas das entidades ao longo das atividades e das filas. Outra alternativa, mais aprimorada, seria a de criar cenários em três dimensões, em que o movimento dos personagens deste cenário seria conduzido pelo simulador. Hoje é possível construir estes cenários de modo integrado ao simulador e até em tempo real utilizando modernos recursos disponíveis nos sistemas operacionais (como o DirectX e OpenGL) e através das modernas placas de vídeo.

Capítulo 9
Conclusões

O desenvolvimento de um ambiente moderno para simulação a eventos discretos utilizando a abordagem envolvendo ao mesmo tempo o Diagrama de Ciclos de Atividades (DCA) e o Método das Três Fases era uma necessidade para os usuários de simulação. O Diagrama de Ciclos de Atividades estrutura os problemas de simulação de modo bastante lógico e de fácil compreensão. O uso desta metodologia era tradicionalmente feito manualmente, sendo inexistente um sistema automático que criasse e executasse o processamento dos modelos em DCA diretamente na tela do computador.

Foi desenvolvido um novo ambiente, denominado SimVisio com esta finalidade. Durante a sua eleboração, vários desenvolvimentos foram realizados visando conferir uma maior facilidade tanto na modelagem quanto no processamento e análise das simulações. A maior contribuição deste trabalho foi proporcionar uma interface simples e intuitiva, sendo de fácil aprendizado de uso. Os diagramas são montados na tela com grande facilidade, utilizando elementos gráficos representativos dos blocos lógicos e de suas ligações. É fácil criar diferentes cenários para a simulação, bastando para isto acionar os blocos do modelo e preencher as suas propriedades em uma janela que se abre sobre eles. O processamento da simulação é também bastante simples, bastando acionar o bloco de comando da simulação para que ela inicie, não havendo a necessidade de código intermediário. Cabe ressaltar que é possível modificar as linhas de código do programa de simulação para que ele se adeque a problemas específicos que fogem à modelagem através dos elementos disponíveis no DCA.

Foram criados vários blocos de modelagem que não existiam no DCA padrão, e que tornaram possível a modelagem de uma série de situações que anteriormente somente poderiam ser simuladas através de mudanças especiais nos códigos escritos dos programas. Os blocos que foram criados foram o de fila para entidades permantentes, chamados também de recursos (FilaRec), o de desvio condicional (Condição), o de inspeção com base em probabilidade de rejeito (Inspeção) e o de busca automática pela menor fila posterior (Auto).

Vale a pena destacar a grande utilidade do uso de um Parser para o processamento de funções matemáticas dentro dos blocos lógicos da simulação.

Através do Parser (na verdade um componente Delphi) tornou-se possível fazer atribuições de valores para alguns campos e atributos das entidades do modelo, sem a necessidade de modificar o código-fonte do programa, mas indiretamente, indicando em alguns campos próprios definidos dentro dos blocos lógicos do modelo em DCA.

Um recurso muito útil foi criado para facilitar a leitura e visualização da lógica do modelo. Trata-se da identificação automática dos ciclos de cada entidade no diagrama. Cada ciclo recebe uma cor diferente, representando a entidade que está sendo representada. Esta cor é definida pelo programa na medida que ele percorre o modelo e identifica um a um cada bloco ou conector pertencente ao modelo.

Paralelamente à interface foi também desenvolvido um sistema de depuração do modelo segundo regras de sintaxe. Esta funcionalidade serviu para facilitar a tarefa de modelagem do usuário, visto que os eventuais erros de construção são comunicados de forma clara, evitando assim enganos e agilizando o processo de desenvolvimento.

O algoritmo de simulação utilizado, bastante conhecido no mundo da simulação, denominado Método das Três Fases, mostrou-se rápido e preciso nos resultados, o que era de se esperar, visto que o código utilizado foi adaptado de sistemas anteriores que já haviam sido bastante testados.

O código do programa utilizou a abordagem orientada a componentes, pequenos módulos de código que possuem regras bem definidas de comportamento e de comunicação entre si. O uso dos componentes foi fundamental para o bom desempenho sistema, em primeiro lugar porque facilitou a transferência das informações do programa de criação do modelo (Visio) para o programa de processamento da simulação (SimVisio), desenvolvido na linguagem Delphi. O uso dos componentes foi bastante útil, sobretudo na fase de programação, pela vantagem de propiciar uma maior rapidez na depuração de erros. Além disso, permitiu a execução de simulações envolvendo uma grande massa de dados com um bom desempenho, tanto pela rapidez de processamento quanto pela ausência de erros de acesso à memória do computador. Outra característica muito importante dos componentes de simulação é a reutilização, que foi amplamente usada no presente ambiente, visto que muitos dos componentes foram adaptados de sistemas de simulação anteriores.

Como resultado final consideramos que o trabalho possui uma série de inovações no campo da simulação e esperamos que ele seja útil para o uso de profissionais e pesquisadores da área.

Trabalhos Futuros

Algumas sugestões podem ser feitas para trabalhos futuros dando seqüência a esta linha de pesquisa, fruto das experiências recolhidas na aplicação deste sistema.

Em primeiro lugar, a utilização de uma plataforma de modelagem mais integrada. Isto poderia ser feita basicamente de duas formas. A primeira, desenvolvendo as ferramentas de processamento da simulação dentro do próprio Visio sem a utilização de compiladores externos, mas somente utilizando a sua linguagem base, o Visual Basic. Outro caminho seria através de compiladores puros como o Delphi, acrescentando-se interpretadores de código para instruções a serem indicadas nos blocos lógicos do modelo. Ambas soluções agilizariam a transferência das informações ente a interface de modelagem e o programa de processamento da simulação.

Em segundo lugar seria interessante a criação de mais elementos no DCA. Por exemplo, verifica-se a necessidade de blocos para agrupar e desagrupar entidades, necessários para alguns modelos específicos.

Outra sugestão de trabalho futuro poderia ser a implementação de modelagem hierárquica. Esta abordagem é especialmente útil para a criação de modelos com um grande número de elementos, onde algumas partes do modelo principal poderiam ser agrupadas em submodelos, sendo que cada submodelo ficaria representado no modelo principal através de um bloco apenas. O acesso ao conteúdo do submodelo poderia então ser feito através de deste bloco representativo, “clicando-se” nele e abrindo uma nova janela exclusiva para esta parte do modelo lógico. A implementação de modelos hierárquicos é possível de ser feita tanto em Visio quanto em Delphi.

Por fim, um bom complemento para este estudo seria um módulo para animação gráfica das entidades do modelo, podendo ser feito em duas ou três dimensões.

Referências Bibliográficas

ANGULO, I. E., 1983. *New facilities for combined simulation modelling*. Tese de Doutorado , Universidade de Lancaster, Lancaster, Inglaterra.

BANKS, J., CARSON, J.S., NELSON, B.L., 1999, *Discrete-Event System Simulation*. 2 ed. New Jersey, Prentice-Hall.

BALCI, O., BERTELRUD A.I., ESTERBROOK C.M. e NANCE R.E. 1998, "Visual Simulation Environment". In: *Proceedings of the 1998 Winter Simulation Conference.* December.

BALMER, D.; PAUL, R.J., 1986 "CASM - The right environment for simulation". *Journal of the Operational Research Society*, v. 37, n. 5 (May), p. 443-452.

CANTÙ, M., 1998, *Dominando o Delphi 3*. São Paulo, Makron Books.

CARVALHO, R.S., 1975, *Cellular simulation*. Tese de Doutorado, Universidade de Lancaster, Lancaster, Inglaterra.

CHEN, G., SZYMANKI, B. K., 2001, "Component-oriented Simulation Architecture: Toward Interoperability and Interchageability ". In: *Proceedings of the 2001 Winter Simulation Conference*, pp. 495-501, December.

CROOKES, J.G., 1985, *Simulation using Turbo Pascal*. Working Paper, Universidade de Lancaster, Lancaster, Inglaterra.

CROOKES, J.G. et al., 1986, "A three-phase simulation system written in Pascal". *Journal of the Operational Research Society*, v. 37, n.6 (June), pp. 603-618.

DAVIES,R., O'KEEFE,R., 1989, *Simulation modelling with Pascal*. Hemel Hempstead, Prentice Hall.

EGE, R.K., 1992, *Programming in an Object-Oriented Environment*, Cambridge, USA, Academic Press.

FUKUNARI, M., CHI Y. e WOLFE P. M., 1998, "JavaBean-based simulation with a decision making bean". In: *Proceedings of the 1998 Winter Simulation Conference.* December.

HEALY, K. J. e KILGORE, R. A., 1998, “Introduction to Silk and Java-based simulation ”. In: *Proceedings of the 1998 Winter Simulation Conference*. December.

HLUPIC, V., 2000, “Simulation Software: An Operational Research Society Survey of Academic and Industrial Users”. In: *Proceedings of the 2000 Winter Simulation Conference.* December.

JOINES, J.A., ROBERTS, S.D., 1998, “Fundamentals of Object-Oriented Simulation”. In: *Proceedings of the 1998 Winter Simulation Conference,* pp. 141-149, Washington, D.C., December.

KACHITVICHYANUKUL, V., HENRIKSE, J.O., PEGDEN, C. D., INGALLS, R. G., SHMEISER, B. W. 2001, “Simulation Environment for the New Millenium (Panel)”. In: *Proceedings of the 2001 Winter Simulation Conference,* pp. 541-547, December.

KAMIGAMI, T., NAKAMURA, N., 1996, “An Object-Oriented Visual Model-Building and Simulation System for FMS Control”, *Simulation*, 67:6 (Dec), pp. 375-385.

KIENBAUM, G., PAUL, R. J., 1994, “H-ACD: Hierarchical Activity Cycle Diagrams for Object-Oriented Simulation Modeling”. In: *Proceedings of the 1994 Winter Simulation Conference.* December.

KIENBAUM, G. 1995. *A Framework for Automatic Simulation Modelling Using an Object-Oriented Approac*”. Tese Ph.D., Brunel University, England.

MARTINEZ, J.C., 2001, “Ezstrobe – General-Purpose Simulation System based on Activity Cycle Diagrams”. In: *Proceedings of the 2001 Winter Simulation Conference,* pp. 1556-1564, December.

MARTINEZ, J.C., e IOANNOU, P.G., 1999, “General-Purpose Systems for Effective Construction Simulation”. *Journal of Construction Engineering and Management.* pp. 265-275 (Jul-Aug).

MILLER, J.A., GE,Y., TAO, J., 1998, “Component-Based Simulation Environments: JSIM as a Case Study Using Java Beans”. In: *Proceedings of the 1998 Winter Simulation Conference*, pp. 373-381, Washington, D.C., December.

ODHABI, H.I., PAUL, R.J., MACREDIE, R.D., 1998, "Developing a Graphical User Interface for Discrete Event Simulation". In: *Proceedings of the 1998 Winter Simulation Conference*, pp. 429-436, Washington, D.C., December.

PIDD, M., 1998, *Computer Simulation in Management Science*. 4 Ed. Chichester, England, John Wiley & Sons.

PIDD, M., OSES, N., BROOKS, R. J. 1999, "Component-based Simulation on the Web? ". In: *Proceedings of the 1999 Winter Simulation Conference*, pp. 1438-1444, December.

PIMENTEL, M., 1989, *Sistema computacional para simulação discreta com opção de uso da amostragem descritiva*, Dissertação de M.Sc., IME, Rio de Janeiro, RJ, Brasil.

PINTO, L. R., 1999, *Metodologia de Análise do Planejamento de Lavra de Minas a Céu Aberto Baseada em Simulação das Operações de Lavra*. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

POOLEY, HUGUES, 1991, "Towards a Standart for Hierarchical Process Oriented Discrete Event Simulation Diagrams. Part II: The Suggested Approach for Flat Models". *Transactions of the Society for Computer Simulation*, 8 (1): 33-41.

ROBERTS, C.A., DESSOUKY, Y.M., "An Overview of Object-Oriented Simulation", *Simulation*, 70:6 (Jun), pp. 359-368.

ROBINSON, S. 1999, "Three sources of Simulation Inaccuracy (and How to Overcome Them". In: *Proceedings of the 1999 Winter Simulation Conference*, pp. 1701-1708, December.

SALIBY, E., 1995. *Simul 3.1 Manual do Usuário*, Coppead/UFRJ, Rio de Janeiro, RJ, Brasil.

SHI, J., 1997, "A Conceptual Activity Cycle-Based Simulation Modeling Method". In: *Proceedings of the 1997 Winter Simulation Conference*, pp. 1127-1133, December.

SOUZA, G. V., 1994, *Ambiente computacional para simulação a eventos discretos, utilizando o paradigma da programação orientada a objetos*, Dissertação de M.Sc., COPPEAD/UFRJ, Rio de Janeiro, RJ, Brasil.

SYSPACK LTD, 1989, *The VS6 user's guide*. London, Syspack.

TAVARES, L.V., OLIVEIRA, R.C., THEMIDO, I.H., CORREIA, F.N. *Investigação Operacional*, Lisboa, McGraw-Hill, 1996.

TOCHER, K. D. , 1963, *The art of simulation*, London, English Universities Press.

ZEIGLER, B.P. 1987, "Multi-facetted Modelling and Discrete Event Simulation ". Academic Press, New York.

Apêndice 1
Definições dos Objetos Básicos da Biblioteca do Simin

TProblema = class(TObject) ***Define o problema a ser simulado***
public
Titulo : Str70; *Titulo dos relatorios*
Duracao : Double; *Duracao das corridas*
Aquec : Double; *Duracao do Aquecimento*
NCorr : Integer; *Numero de corridas do problema*
Temp : double; *Horario Corrente da simulacao*
Corrida : Integer; *Numero da corrida corrente*
FimCorr : double; *Horario de termino da corrida corrente*
NumProxB : Integer; *Nr. do proximo evento B*
NAtiv : Integer; *Nr. de atv's*
NHist : Integer; *Nr. de histogramas*
NFila : Integer; *Nr. de filas*
Constructor Create;
Procedure DefineFimCorrida; *FimCorr = Duracao + Aquec*
Function Fim : Boolean; *Fim = True => Final de simulacao*
Procedure IncrementaCorrida;
Procedure ProximaCorrida; *Aquec = 0; FimCorr = Temp + Duracao*
Procedure ReinicializaParametros; *Temp = 0; Corrida = 1; FimCorr = 0*
Procedure ProximoTempo; *Avanca relogio ate o prox termino de atv*
end;

TDistrib = class(TComponent) ***Distribuições de probabilidades***
private
FTipo : TTipo_Dist;
FParam_1 : Double;
FParam_2 : Double;
FParam_3 : Double;
FSemente : Integer;
FNV : Integer; *Número de valores da distribucao empirica*
FXX, FYY : TDistDados; *Dados das distribuicoes empiricas*
procedure SetTipo(Valor : TTipo_Dist);
procedure SetSemente(Valor : Integer);
procedure SetParam1(Valor : Double);
procedure SetParam2(Valor : Double);
procedure SetParam3(Valor : Double);
public
Amostra : Double;
Constructor Create(Owner: TComponent); override;
procedure Faz_Amostragem;
Destructor Destroy; override;

published

```
 Property Tipo : TTipo_Dist read FTipo write SetTipo default Dist_Normal;
 Property Semente : Integer read FSemente write SetSemente;
 Property Param_1 : Double read FParam_1 write SetParam1;
 Property Param_2 : Double read FParam_2 write SetParam2;
 Property Param_3 : Double read FParam_3 write SetParam3;
 Property Valores : Integer read FNV write FNV;
 Property X_01 : Double read Fxx[1] write Fxx[1];
 Property X_02 : Double read Fxx[2] write Fxx[2];
 Property X_03 : Double read Fxx[3] write Fxx[3];
 Property X_04 : Double read Fxx[4] write Fxx[4];
 Property X_05 : Double read Fxx[5] write Fxx[5];
 Property X_06 : Double read Fxx[6] write Fxx[6];
 Property X_07 : Double read Fxx[7] write Fxx[7];
 Property X_08 : Double read Fxx[8] write Fxx[8];
 Property X_09 : Double read Fxx[9] write Fxx[9];
 Property X_10 : Double read Fxx[10] write Fxx[10];
 Property X_11 : Double read Fxx[11] write Fxx[11];
 Property X_12 : Double read Fxx[12] write Fxx[12];
 Property X_13 : Double read Fxx[13] write Fxx[13];
 Property X_14 : Double read Fxx[15] write Fxx[14];
 Property X_15 : Double read Fxx[15] write Fxx[15];
 Property X_16 : Double read Fxx[16] write Fxx[16];
 Property X_17 : Double read Fxx[17] write Fxx[17];
 Property X_18 : Double read Fxx[18] write Fxx[18];
 Property X_19 : Double read Fxx[19] write Fxx[19];
 Property X_20 : Double read Fxx[20] write Fxx[20];
 Property Y_01 : Double read FYY[1] write FYY[1];
 Property Y_02 : Double read FYY[2] write FYY[2];
 Property Y_03 : Double read FYY[3] write FYY[3];
 Property Y_04 : Double read FYY[4] write FYY[4];
 Property Y_05 : Double read FYY[5] write FYY[5];
 Property Y_06 : Double read FYY[6] write FYY[6];
 Property Y_07 : Double read FYY[7] write FYY[7];
 Property Y_08 : Double read FYY[8] write FYY[8];
 Property Y_09 : Double read FYY[9] write FYY[9];
 Property Y_10 : Double read FYY[10] write FYY[10];
 Property Y_11 : Double read FYY[11] write FYY[11];
 Property Y_12 : Double read FYY[12] write FYY[12];
 Property Y_13 : Double read FYY[13] write FYY[13];
 Property Y_14 : Double read FYY[15] write FYY[14];
 Property Y_15 : Double read FYY[15] write FYY[15];
 Property Y_16 : Double read FYY[16] write FYY[16];
 Property Y_17 : Double read FYY[17] write FYY[17];
 Property Y_18 : Double read FYY[18] write FYY[18];
```

```
 Property Y_19 : Double read FYY[19] write FYY[19];
 Property Y_20 : Double read FYY[20] write FYY[20];
end;

TEntidade = class(TComponent)  Classe de entidades
private
public
 Tipo : TipoEntidade;
 Quant : Integer;                    Nr. de entidades daquela classe
 Atr: TList;                         Lista de classe de atributos da classe de entidade
 Lista_Dados_Corrida : TList;        Lista p/ acumular dados estatisticos das entidades
 Constructor Create(Owner: TComponent); override;
 Procedure ZeraQuantidade;
 Procedure IncrementaQuantidade;
 Procedure ComputaTempoUso(Cor : Integer; Tempo : Double);
 Procedure CalculaDados;
 Destructor Destroy; override;
published
end;
TAtributo = class(TComponent)   Classe de atributos
private
 FEntidade : TEntidade;
 FIdentificador : Str50;
 Procedure SetEntidade(E : TEntidade);
public
 Destructor Destroy; override;
published
 Property Identificador : Str50 read FIdentificador write FIdentificador;
 Property Entidade : TEntidade read FEntidade write SetEntidade;
end;

TAtribEntEmAtv = class(TObject)  Atributos propriamente ditos
public
 Atributo : TAtributo;          Classe de atributos a que pertence
 Valor : Double;                Valor do atributo
 Constructor Create(iatrib:TAtributo);
end;

TEntidadeAtiv = class(TObject)  Entidades propriamente ditas
public
 Ent: TEntidade;          Classe de entidade a que pertence
 AtrL: TList;             Lista de Atributos da entidade
 NumEventoB, TEsp:double;
 NumEnt:Integer;
 Prioridade : Byte;
```

```
 Constructor Create(IEnt : TEntidade; iprior: byte);
 Procedure CriaAtributos;
 Procedure EsvaziaAtributos;
 Function AvaliaAtributo(inome:Str50): double;
 Procedure DestroiListaAtrib;
 Procedure DefineValorAtributo (NomeAt : Str50; ValorAt: double);
 Function AchaAtributo (NomeAt: Str50): TAtributo;
end;
```

THistograma = class(TComponent) ***Histogramas***

```
private
 FHTipo : HistTipo;
 FLargura : Double;
 FBase : Double;
public
 TFlag2 : Double;
 Lista_Dados_Corrida : TList; Lista dos objetos TDado_Hist - Globvar
 Constructor Create(Owner : TComponent);override;
 Procedure AcumulaDadosCel (Dado : double; Cor : Integer);
 Procedure CalculaDados;
 Destructor Destroy; override;
published
 Property Htipo : HistTipo read FHtipo write FHTipo;
 Property Largura : Double read FLargura write FLargura;
 Property Base : Double read FBase write FBase;
end;
```

TFila = class(TComponent) ***Filas***

```
private
 FEnt: TEntidade;
 FPrimeiro, FUltimo: Word;
 FPrioridade: Byte;
 FHistTamanho, FHistEspera: THistograma;
 Procedure SetEnt(E : TEntidade);
public
 Soma, TFlag : Double;
 Componentes: TList;
 Nr_Evento_B : Integer;              Nr. do Evento B associado a fila
 Constructor Create(Owner:TComponent); override;
 Procedure Esvazia;
 Procedure Enche;
 Procedure VerificaTempoEspera (EntAtiv : TEntidadeAtiv);
 Function Tira (Posicao:Boolean) : TEntidadeAtiv; virtual;
```

```
 Function RetiraPorAtributo (AtribNome: Str50; Maior: Boolean): TEntidadeAtiv;
 Procedure MarcaTempo(EntAt : TEntidadeAtiv);
 Procedure Coloca (var EntAtiv : TEntidadeAtiv; Posicao : Boolean);
 Procedure PriorMax (MaxPrior : Byte; Index : Integer);
 Function EnesimaEnt (N: Integer): TEntidadeAtiv;
 Function AchaEnt (EntALocalizar: TEntidadeAtiv): Boolean;
 Destructor Destroy; override;
published
 Property Ent : TEntidade read FEnt write SetEnt;
 Property Primeiro : Word read FPrimeiro write FPrimeiro;
 Property Ultimo : Word read FUltimo write FUltimo;
 Property Prioridade : Byte read FPrioridade write FPrioridade;
 Property HistTamanho : THistograma read FHistTamanho write FHistTamanho;
 Property HistEspera : THistograma read FHistEspera write FHistEspera;
end;

TFilaPri = class(TFila) Filas com prioridade
public
 Function Tira(Posicao:Boolean) : TEntidadeAtiv; override;
end;

TFonte = class(TComponent) Fontes/Sumidouros
private
 FEnt: TEntidade;
 FPrioridade: Byte;
 Procedure SetEnt(E : TEntidade);
public
 Cont: Integer;
 Nr_Evento_B : Integer;          Nr. do Evento B associado a fonte
 Constructor Create(Owner : TComponent); override;
 Procedure ZeraContador;
 Function Tira(Prior : Byte) : TEntidadeAtiv;
 Destructor Destroy; override;
published
 Property Ent : TEntidade read FEnt write SetEnt;
 Property Prioridade : Byte read FPrioridade write FPrioridade;
end;


TEvento_B = class(Tobject)   Ligacao entre o nr. do evento B e a Fila de destino
   Numero_Evento : Integer;
   Atv_Mineracao : TProcesso;
   Fim_Condicional : Boolean;      Avisa se a Atv tem fim condicional
   Fila : TFila;                   Fila para onde irá a Ent apos termino da Atv
```

```
  Nome_Ent : String;          Nome da Ent que esta saindo da Atv
  Nome_Atv : String;          Nome da Atv que esta a terminar
end;

TAtividade = class(TComponent) Atividades
private
 FHist: THistograma;
 FDist: TDistrib;
 FPrioridade : Integer;        Prioridade de execução da atividade
 FIni_Condicional,FFim_Condicional : Boolean;     Indicadores de Atv Condicionais
 FFilaPre : array [1..10] of TFila;
 FFilaPos : array [1..10] of TFila;
 FFonte : array [1..10] of TFonte;
 procedure SetDist (Valor : TDistrib);
 procedure SetFilaPre_1 (Valor : TFila);
 procedure SetFilaPre_2 (Valor : TFila);
 procedure SetFilaPre_3 (Valor : TFila);
 procedure SetFilaPre_4 (Valor : TFila);
 procedure SetFilaPre_5 (Valor : TFila);
 procedure SetFilaPre_6 (Valor : TFila);
 procedure SetFilaPre_7 (Valor : TFila);
 procedure SetFilaPre_8 (Valor : TFila);
 procedure SetFilaPre_9 (Valor : TFila);
 procedure SetFilaPre_10 (Valor : TFila);
 procedure SetFilaPos_1 (Valor : TFila);
 procedure SetFilaPos_2 (Valor : TFila);
 procedure SetFilaPos_3 (Valor : TFila);
 procedure SetFilaPos_4 (Valor : TFila);
 procedure SetFilaPos_5 (Valor : TFila);
 procedure SetFilaPos_6 (Valor : TFila);
 procedure SetFilaPos_7 (Valor : TFila);
 procedure SetFilaPos_8 (Valor : TFila);
 procedure SetFilaPos_9 (Valor : TFila);
 procedure SetFilaPos_10 (Valor : TFila);
 procedure SetFonte_1 (Valor : TFonte);
 procedure SetFonte_2 (Valor : TFonte);
 procedure SetFonte_3 (Valor : TFonte);
 procedure SetFonte_4 (Valor : TFonte);
 procedure SetFonte_5 (Valor : TFonte);
 procedure SetFonte_6 (Valor : TFonte);
 procedure SetFonte_7 (Valor : TFonte);
 procedure SetFonte_8 (Valor : TFonte);
 procedure SetFonte_9 (Valor : TFonte);
 procedure SetFonte_10 (Valor : TFonte);
public
```

```
 Prox_Chegada : Double;      Armazena qdo sera a prox chegada p/ atv's de chegada
 Filas_pre, Filas_pos, Fontes : TList;      Lista de filas Pre, Pos Atv e Fontes Pre
 Lista_Dados_Corrida : TList;      Lista p/ acumular dados estatisticos das atv's
 Tipo_Atv : TTipo_Atividade;
 Constructor Create(Owner:TComponent);override;
 Function Duracao_Atv : Double;
 Function Pode_Iniciar(Tempo : Double) : Boolean;
 Procedure Programa(Cor : Integer);
 Procedure Tira_Da_Lista_Fila_Pre(Fila:TFila);
 Procedure Tira_Da_Lista_Fila_Pos(Fila:TFila);
 Procedure Coloca_na_Lista_de_Filas(Fila : TFila; Tipo_lista : Str3);
 Procedure Tira_Da_Lista_Fontes(Fonte:TFonte);
 Procedure Coloca_na_Lista_de_Fontes(Fonte : TFonte);
 Procedure CalculaDados;
 Destructor Destroy; override;
published
 Property Hist : THistograma read FHist write FHist;
 Property Distribuicao : TDistrib read FDist write SetDist;
 Property Prioridade : Integer read FPrioridade write FPrioridade;
 Property Ini_Condicional : Boolean read FIni_Condicional write FIni_Condicional;
 Property Fim_Condicional : Boolean read FFim_Condicional write
FFim_Condicional;
 Property FilaAnt_01 : TFila read FFilaPre[1] write SetFilaPre_1;
 Property FilaAnt_02 : TFila read FFilaPre[2] write SetFilaPre_2;
 Property FilaAnt_03 : TFila read FFilaPre[3] write SetFilaPre_3;
 Property FilaAnt_04 : TFila read FFilaPre[4] write SetFilaPre_4;
 Property FilaAnt_05 : TFila read FFilaPre[5] write SetFilaPre_5;
 Property FilaAnt_06 : TFila read FFilaPre[6] write SetFilaPre_6;
 Property FilaAnt_07 : TFila read FFilaPre[7] write SetFilaPre_7;
 Property FilaAnt_08 : TFila read FFilaPre[8] write SetFilaPre_8;
 Property FilaAnt_09 : TFila read FFilaPre[9] write SetFilaPre_9;
 Property FilaAnt_10 : TFila read FFilaPre[10] write SetFilaPre_10;
 Property FilaPos_01 : TFila read FFilaPos[1] write SetFilaPos_1;
 Property FilaPos_02 : TFila read FFilaPos[2] write SetFilaPos_2;
 Property FilaPos_03 : TFila read FFilaPos[3] write SetFilaPos_3;
 Property FilaPos_04 : TFila read FFilaPos[4] write SetFilaPos_4;
 Property FilaPos_05 : TFila read FFilaPos[5] write SetFilaPos_5;
 Property FilaPos_06 : TFila read FFilaPos[6] write SetFilaPos_6;
 Property FilaPos_07 : TFila read FFilaPos[7] write SetFilaPos_7;
 Property FilaPos_08 : TFila read FFilaPos[8] write SetFilaPos_8;
 Property FilaPos_09 : TFila read FFilaPos[9] write SetFilaPos_9;
 Property FilaPos_10 : TFila read FFilaPos[10] write SetFilaPos_10;
 Property Fonte_01 : TFonte read FFonte[1] write SetFonte_1;
 Property Fonte_02 : TFonte read FFonte[2] write SetFonte_2;
 Property Fonte_03 : TFonte read FFonte[3] write SetFonte_3;
```

```
 Property Fonte_04 : TFonte read FFonte[4] write SetFonte_4;
 Property Fonte_05 : TFonte read FFonte[5] write SetFonte_5;
 Property Fonte_06 : TFonte read FFonte[6] write SetFonte_6;
 Property Fonte_07 : TFonte read FFonte[7] write SetFonte_7;
 Property Fonte_08 : TFonte read FFonte[8] write SetFonte_8;
 Property Fonte_09 : TFonte read FFonte[9] write SetFonte_9;
 Property Fonte_10 : TFonte read FFonte[10] write SetFonte_10;
end;
```

TAtivExec = class(TObject) ***Atividades propriamente ditas***

```
public
 Ativ: TAtividade;
 TInicio: double;
 TFim: double;
 ListaEnt: TList;   {Lista de EntidadeAtiv}
 Constructor Create(iativ : TAtividade; itinicio: Double;itfim: double);
 Function TempoExecucao : double;
end;
```

Variáveis Globais

```
AtvEmPreparo, AtvCorrente : TAtivExec;
Reinicio, Termino : TAtividade;
Sistema, EntdoTermino, EntdoReinicio : TEntidade;
EntAtivdoTermino, EntAtivdoReinicio, EntAtivdoSist, EntCorrente: TEntidadeAtiv;
ProbSim : TProblema;
Evento_B : TEvento_B;
Lista_Eventos_B : TList;
Cont_Evento_B, Maior_Evento_B_Automatico : Integer;
Simulacao_Executada, Simulacao_Em_Execucao, Modelo_Checado : Boolean;
```

Apêndice 2
Definição das Constantes, Tipos, Variáveis Globais, Funções e Procedimentos das Interfaces das Unidades da Biblioteca do SimVisio

UNIDADE DE COMPONENTES DE SIMULAÇÃO (Compnt.pas)
Rotinas do executivo de simulação

Variáveis Globais
AtvEmPreparo, AtvCorrente : TAtivExec;
Reinicio, Termino : TAtividade;
Sistema, EntdoTermino, EntdoReinicio : TEntidade;
EntAtivdoTermino, EntAtivdoReinicio, EntAtivdoSist, EntCorrente: TEntidadeAtiv;
ProbSim : TProblema;
Evento_B : TEvento_B;
Lista_Eventos_B : TList;
Cont_Evento_B, Maior_Evento_B_Automatico : Integer;
Simulacao_Executada, Simulacao_Em_Execucao, Modelo_Checado : Boolean;

TProblema = class(TObject) ***Define o problema a ser simulado***
public
Titulo : Str70; *Titulo dos relatorios*
Duracao : Double; *Duracao das corridas*
Aquec : Double; *Duracao do Aquecimento*
NCorr : Integer; *Numero de corridas do problema*
Temp : double; *Horario Corrente da simulacao*
Corrida : Integer; *Numero da corrida corrente*
FimCorr : double; *Horario de termino da corrida corrente*
NumProxB : Integer; *Nr. do proximo evento B*
NAtiv : Integer; *Nr. de atv's*
NHist : Integer; *Nr. de histogramas*
NFila : Integer; *Nr. de filas*
Constructor Create;
Procedure DefineFimCorrida; *FimCorr = Duracao + Aquec*
Function Fim : Boolean; *Fim = True => Final de simulacao*
Procedure IncrementaCorrida;
Procedure ProximaCorrida; *Aquec = 0; FimCorr = Temp + Duracao*
Procedure ReinicializaParametros; *Temp = 0; Corrida = 1; FimCorr = 0*
Procedure ProximoTempo; *Avanca relogio ate o prox termino de atv*
end;

TDistrib = class(TComponent) ***Distribuições de probabilidades***

```
private
  FTipo : TTipo_Dist;
  FParam_1 : Double;
  FParam_2 : Double;
  FParam_3 : Double;
  FSemente : Integer;
  FNV : Integer;          Número de valores da distribucao empirica
  FXX, FYY : TDistDados;  Dados das distribuicoes empiricas
  procedure SetTipo(Valor : TTipo_Dist);
  procedure SetSemente(Valor : Integer);
  procedure SetParam1(Valor : Double);
  procedure SetParam2(Valor : Double);
  procedure SetParam3(Valor : Double);
public
 Amostra : Double;
 Constructor Create(Owner: TComponent); override;
 procedure Faz_Amostragem;
 Destructor Destroy; override;
published
 Property Tipo : TTipo_Dist read FTipo write SetTipo default Dist_Normal;
 Property Semente : Integer read FSemente write SetSemente;
 Property Param_1 : Double read FParam_1 write SetParam1;
 Property Param_2 : Double read FParam_2 write SetParam2;
 Property Param_3 : Double read FParam_3 write SetParam3;
 Property Valores : Integer read FNV write FNV;
 Property X_01 a X_20 : Double read Fxx[1] write Fxx[1];
 Property Y_01 a Y_20 : Double read FYY[1] write FYY[1];
end;

TEntidade = class(TComponent)  Classe de entidades
private
public
 Tipo : TipoEntidade;
 Quant : Integer;                Nr. de entidades daquela classe
 Atr: TList;                     Lista de classe de atributos da classe de entidade
 Lista_Dados_Corrida : TList;    Lista p/ acumular dados estatisticos das entidades
 Constructor Create(Owner: TComponent); override;
 Procedure ZeraQuantidade;
 Procedure IncrementaQuantidade;
 Procedure ComputaTempoUso(Cor : Integer; Tempo : Double);
 Procedure CalculaDados;
 Destructor Destroy; override;
published
end;
```

TAtributo = class(TComponent) ***Classe de atributos***
private
FEntidade : TEntidade;
FIdentificador : Str50;
Procedure SetEntidade(E : TEntidade);
public
Destructor Destroy; override;
published
Property Identificador : Str50 read FIdentificador write FIdentificador;
Property Entidade : TEntidade read FEntidade write SetEntidade;
end;

TAtribEntEmAtv = class(TObject) ***Atributos propriamente ditos***
public
Atributo : TAtributo; *Classe de atributos a que pertence*
Valor : Double; *Valor do atributo*
Constructor Create(iatrib:TAtributo);
end;

TEntidadeAtiv = class(TObject) ***Entidades propriamente ditas***
public
Ent: TEntidade; *Classe de entidade a que pertence*
AtrL: TList; *Lista de Atributos da entidade*
NumEventoB, TEsp:double;
NumEnt:Integer;
Prioridade : Byte;
Constructor Create(IEnt : TEntidade; iprior: byte);
Procedure CriaAtributos;
Procedure EsvaziaAtributos;
Function AvaliaAtributo(inome:Str50): double;
Procedure DestroiListaAtrib;
Procedure DefineValorAtributo (NomeAt : Str50; ValorAt: double);
Function AchaAtributo (NomeAt: Str50): TAtributo;
end;

THistograma = class(TComponent) ***Histogramas***
private
FHTipo : HistTipo;
FLargura : Double;
FBase : Double;
public
TFlag2 : Double;
Lista_Dados_Corrida : TList; Lista dos objetos TDado_Hist - Globvar
Constructor Create(Owner : TComponent);override;
Procedure AcumulaDadosCel (Dado : double; Cor : Integer);

```
 Procedure CalculaDados;
 Destructor Destroy; override;
published
 Property Htipo : HistTipo read FHtipo write FHTipo;
 Property Largura : Double read FLargura write FLargura;
 Property Base : Double read FBase write FBase;
end;

TFila = class(TComponent) Filas
private
 FEnt: TEntidade;
 FPrimeiro, FUltimo: Word;
 FPrioridade: Byte;
 FHistTamanho, FHistEspera: THistograma;
 Procedure SetEnt(E : TEntidade);
public
 Soma, TFlag : Double;
 Componentes: TList;
 Nr_Evento_B : Integer;              Nr. do Evento B associado a fila
 Constructor Create(Owner:TComponent); override;
 Procedure Esvazia;
 Procedure Enche;
 Procedure VerificaTempoEspera (EntAtiv : TEntidadeAtiv);
 Function Tira (Posicao:Boolean) : TEntidadeAtiv; virtual;
 Function RetiraPorAtributo (AtribNome: Str50; Maior: Boolean): TEntidadeAtiv;
 Procedure MarcaTempo(EntAt : TEntidadeAtiv);
 Procedure Coloca (var EntAtiv : TEntidadeAtiv; Posicao : Boolean);
 Procedure PriorMax (MaxPrior : Byte; Index : Integer);
 Function EnesimaEnt (N: Integer): TEntidadeAtiv;
 Function AchaEnt (EntALocalizar: TEntidadeAtiv): Boolean;
 Destructor Destroy; override;
published
 Property Ent : TEntidade read FEnt write SetEnt;
 Property Primeiro : Word read FPrimeiro write FPrimeiro;
 Property Ultimo : Word read FUltimo write FUltimo;
 Property Prioridade : Byte read FPrioridade write FPrioridade;
 Property HistTamanho : THistograma read FHistTamanho write FHistTamanho;
 Property HistEspera : THistograma read FHistEspera write FHistEspera;
end;

TFonte = class(TComponent) Fontes/Sumidouros
private
 FEnt: TEntidade;
 FPrioridade: Byte;
 Procedure SetEnt(E : TEntidade);
```

```
public
 Cont: Integer;
 Nr_Evento_B : Integer;              Nr. do Evento B associado a fonte
 Constructor Create(Owner : TComponent); override;
 Procedure ZeraContador;
 Function Tira(Prior : Byte) : TEntidadeAtiv;
 Destructor Destroy; override;
published
 Property Ent : TEntidade read FEnt write SetEnt;
 Property Prioridade : Byte read FPrioridade write FPrioridade;
end;

TEvento_B = class(Tobject)   Ligacao entre o nr. do evento B e a Fila de destino
   Numero_Evento : Integer;
   Atv_Mineracao : TProcesso;
   Fim_Condicional : Boolean;       Avisa se a Atv tem fim condicional
   Fila : TFila;                    Fila para onde irá a Ent apos termino da Atv
   Nome_Ent : String;               Nome da Ent que esta saindo da Atv
   Nome_Atv : String;               Nome da Atv que esta a terminar
end;

TAtividade = class(TComponent) Atividades
private
 FHist: THistograma;
 FDist: TDistrib;
 FPrioridade : Integer;          Prioridade de execução da atividade
 FIni_Condicional,FFim_Condicional : Boolean;       Indicadores de Atv Condicionais
 FFilaPre : array [1..10] of TFila;
 FFilaPos : array [1..10] of TFila;
 FFonte : array [1..10] of TFonte;
 procedure SetDist (Valor : TDistrib);
 procedure SetFilaPre_1 a SetFilaPre_10 (Valor : TFila);
 procedure SetFilaPos_1 a SetFilaPos_10 (Valor : TFila);
 procedure SetFonte_1  a SetFonte_10 (Valor : TFonte);
public
 Prox_Chegada : Double;      Armazena qdo sera a prox chegada p/ atv's de chegada
 Filas_pre, Filas_pos, Fontes : TList;       Lista de filas Pre, Pos Atv e Fontes Pre
 Lista_Dados_Corrida : TList;       Lista p/ acumular dados estatisticos das atv's
 Tipo_Atv : TTipo_Atividade;
 Constructor Create(Owner:TComponent);override;
 Function Duracao_Atv : Double;
 Function Pode_Iniciar(Tempo : Double) : Boolean;
 Procedure Programa(Cor : Integer);
 Procedure Tira_Da_Lista_Fila_Pre(Fila:TFila);
 Procedure Tira_Da_Lista_Fila_Pos(Fila:TFila);
```

```
 Procedure Coloca_na_Lista_de_Filas(Fila : TFila; Tipo_lista : Str3);
 Procedure Tira_Da_Lista_Fontes(Fonte:TFonte);
 Procedure Coloca_na_Lista_de_Fontes(Fonte : TFonte);
 Procedure CalculaDados;
 Destructor Destroy; override;
published
 Property Hist : THistograma read FHist write FHist;
 Property Distribuicao : TDistrib read FDist write SetDist;
 Property Prioridade : Integer read FPrioridade write FPrioridade;
 Property Ini_Condicional : Boolean read FIni_Condicional write FIni_Condicional;
 Property Fim_Condicional : Boolean read FFim_Condicional write
FFim_Condicional;
 Property FilaAnt_01 a FilaAnt_10 : TFila read FFilaPre[10] write SetFilaPre_10;
 Property FilaPos_01 a FilaPos_10 : TFila read FFilaPos[10] write SetFilaPos_10;
 Property Fonte_01 a Fonte_10 : TFonte read FFonte[10] write SetFonte_10;
end;

TAtivExec = class(TObject) Atividades propriamente ditas
public
 Ativ: TAtividade;
 TInicio: double;
 TFim: double;
 ListaEnt: TList;   {Lista de EntidadeAtiv}
 Constructor Create(iativ : TAtividade; itinicio: Double;itfim: double);
 Function TempoExecucao : double;
end;
```

UNIDADE AMOSTRA

```
Rotinas para Amostragem Aleatória Simples
Tipos:
Seeds = 1..NumSementes;
TDistDados = Array[1..DimensD] Of Double;
TDist_Emp = Class(Tobject)
    X,Y: TDistDados;
    Semente : Seeds;
End;
Variáveis Globais:
SementeOriginal, Semente : Array[Seeds] of Integer;
Mult,A,M : Integer;
Funções:
Function Rnd (S:Seeds): Double;
Function GeraContinua (Dist: TDist_Emp; S: Seeds): Double;
Function GeraDiscreta (Dist: TDist_Emp; S: Seeds): Double;
Function Normal (M,DP: Double; S: Seeds): Double;
```

Function NegExp (M: Double; S: Seeds): Double;
Function WeiBull (A,B: Double; S:Seeds): Double;
Function Poisson (M: Double; S: Seeds): Double;
Function Erlang (K: Integer; M: Double; S: Seeds): Double;
Function Triangular (A,B,C: Double; S: Seeds): Double;
Function Uniforme (A,B: Double; S: Seeds): Double;
Function InteiraUniforme (A,B: Double; S:Seeds): Double;
Function Bernoulli (P: Double; S: Seeds): Double;
Function LogNormal (M,DP: Double; S: Seeds): Double;
Function Gama (Beta,Alfa: Double; S: Seeds): Double;
Function Beta(A,B: Integer; S: Seeds): Double;
Procedimentos:
Procedure Continua (Var Dist: TDist_Emp; NV: Integer; XX,YY:TDistDados);
Procedure Discreta (Var Dist: TDist_Emp; NV: Integer; XX,YY: TDistDados);
Procedure InicioAmost;

UNIDADE AUXILIAR

Rotinas auxiliares
Funções:
Function MinimoDe(A,B: Double) : Double;
Function MaximoDe(A,B: Double) : Double;

Procedimento:

Procedure Deleta_Dados_Lista(Lista : TList);

UNIDADE CONTEMPO

Rotinas de controle das atividades em execução

Função
Function Primeiro : TAtivExec;

Procedimentos:
Procedure Insere(NovaAtv : TAtivexec);
Procedure ReatualizaEnt(TempoSim : Double);
Procedure FimCorrida;

UNIDADE ESTATISTICA

Rotinas estatísticas

Procedimentos:

Procedure Cria_Dados_Estatistica;
Procedure Reinicia_Dados_Estatistica;
Procedure Calcula_Dados_Estatistica;

UNIDADE GERENCIADOR

Rotinas de gerenciamento da simulação

Procedimentos:
Procedure Gera_Lista_Eventos;
Procedure Ordena_ListaAtv;
Procedure AtravesCEventos;
Procedure ChamaBEventos;
Procedure Atv_Ent_Fila_Evento_B
(NumEvento : Integer; Var NomeAtv, NomeEnt : String; Fil : TFila);

UNIDADE GLOBVAR

Variáveis globais genéricas

Constantes:
NumSementes = 20;
DimensD = 20;
Atras = True;
Frente = False;
Maior = False;
Menor = True;

Tipos:
str1 = string[1];
str2 = string[2];
str3 = string[3];
str4 = string[4];
str5 = string[5];
str6 = string[6];
str7 = string[7];
str8 = string[8];
str9 = string[9];
str10 = string[10];
str11 = string[11];
str12 = string[12];
str15 = string[15];
str20 = string[20];
str25 = string[25];

```
str50 = string[50];
str70 = string[70];
str100 = string[100];
str128 = string[128];

TControle_Video=class(TThread)
private
protected
  procedure execute; override;
  Procedure Envia_Dados_Tela_Inf;
public
  Rodando : Boolean;
  Corrida : Integer;
  Tempo, Duracao : Double;
  Nome_Atv : String;
end;

HistTipo = (SIMPLES,INTEGRADO);

TDado_Hist = class(TObject)
    Corrida : Integer;
    Cel_Hist : Array [0..16] of double;
    Cont, Minimo, Maximo,  Media, Variancia, DP, Soma, SomaQdr :Double;
end;

TipoEntidade = (PERMANENTE, TEMPORARIA);

TipoColeta = (TESPERA, TAMANHO);

TTipo_Dist = (Dist_Normal,Dist_NegExp,Dist_Weibull,Dist_Poisson,Dist_Erlang,
          Dist_Triangular,Dist_Uniforme,Dist_Int_Uniforme,Dist_Bernoulli,
          Dist_LogNormal,Dist_Gama,Dist_Beta,Dist_Emp_Disc,Dist_Emp_Cont);

TDado_Atv = class(TObject)
    Corrida,Cont : Integer;
    Media, Variancia, DP, Soma, SomaQdr : Double;
end;

TDado_Ent = class(TObject)
    Corrida : Integer;
    TUso, Utilizacao : Double;
end;

TTipo_Atividade = (BASICA, MINERACAO);
```

Variáveis:

ListaEnt, ListaAtv, ListaHist, ListaFila, ListaFonte, ListaAtvExec : Tlist;
Erro : Boolean;
TempoAtv : Double;
VDuracao, VAquecimento : Double;
VNumCor : Integer;
VVideo, VPasso_a_Passo : String[1];
VTitulo : Str70;
VNome : Str8;
VCorrida_Ant : Integer;
Controle_Video : TControle_Video;
Prox_Passo, Parar_Simulacao : Boolean;
Numero_Evento_B : Integer;
Lista_Val_Graf_Hist : TList;

UNIDADE ROTINAS

Rotinas de uso geral de simulação

Funções:

Function TiraDaFonte(Font:TFonte; prior:byte): TEntidadeAtiv;
Function TiraDaFila (Pos : Boolean; Fil : TFila) : TEntidadeAtiv;
Function RetiraEntAtributo(Fil:TFila; Atrib:Str50; MaiMen:Boolean): TEntidadeAtiv;
Function PrimeiroDaFila(Fil: TFila) : TEntidadeAtiv;
Function AvaliaAtributo(EntAtiv: TEntidadeAtiv; Nome: Str50) : Double;
Function MaiorPrioridade(Fil: TFila): Byte;
Function AvaliaPrioridade(EntAtiv: TEntidadeAtiv): Byte;
Function TempoCorrente : Boolean;
Function ProximoEventoB :Boolean;
Function NumeroProxB : Integer;

Procedimentos:
Procedure InicioVars;
Procedure DefineCont(Var Font: TFonte; Quant: Integer);
Procedure DefineSistEnt (Ent : TEntidade);
Procedure EncheFila(Fil:TFila);
Procedure PreparaTermino (NumB:Integer; Dono:TComponent);
Function TamanhoFila(Fil: TFila) : Integer;
Procedure ProgramaAtv (Ativ: TAtividade);
Procedure PreparaB (NumB : Integer; Ent : TEntidadeAtiv);
Procedure FimProgramaAtv;
Procedure AcumulaDadosCel(Hist : THistograma; Valor : Double; Cor :Integer);
Procedure PreparaReinicio(NumB:Integer; Dono:TComponent);
Procedure Destroi (EntAtv : TEntidadeAtiv);
Procedure DefValAtributo(EntAtiv:TEntidadeAtiv; Nome:Str50; Valor:Double);

Procedure ColocaNaFila(var EntAtv:TEntidadeAtiv; Pos:Boolean; Fil:TFila);
Procedure ColocaNaFilaPri(var EntAtv:TEntidadeAtiv; Pos:Boolean; Var Fil:TFilaPri);
Procedure FimAquecimento;
Procedure ProximaCorrida(Dono : TComponent);
Procedure EsvaziaFilasEntTemporarias;

UNIDADE UTIL

Rotinas de controle do processo de simulação e checagem do modelo

Procedimentos:
Procedure Simula(Dono: TComponent);
Procedure Checar_Modelo

Apêndice 3
Definição dos Campos, Tipos, Valores Default e Prompt dos Shapes do Painel DCA da Interface Visio

Shape Fila

N	Nome do Campo	Tipo/ Valores	Default	Prompt
1	"Inicial"	Inteiro a partir de 0	0 (zero)	"Número de entidades na fila no início da simulação".
2	"Prioridade"	Inteiro a partir de 1	1	"Indica a prioridade sobre outras filas, quando necessário. Maior=1".
3	"Disciplina"	FIFO LIFO PorPrioridade PorAtributo(Nome)	FIFO	"Disciplina de entrada e saída na fila. FIFO - first in first out, LIFO - last in first out e PorAtributo - ordem por atributo"
4	"Capacidade"	Infinita ou Inteiro a partir de 1	Infinita	"Capacidade da fila. Pode ser Infinita ou ter limite (número de 1 em diante)"
5	"ExcedeCapacidade"	BloqueiaAtividade ou SaiDoSistema	Bloqueia Atividade	"Indica o que fazer com a entidade que chegar com a fila que estiver acima do limite da capacidade"
6	"Histograma Tamanho"	Sim, Não ou ArquivoTxt	Sim	"Indica se deve ser criado o Histograma do Tamanho da Fila".
7	"Hist Tam - Base"	Inteiro	0.5	"Marca o início do Histograma".
8	"Hist Tam - Largura"	Inteiro a partir de 1	1	"Largura de cada faixa do histograma de tamanho da fila".
9	"Histograma Espera"	Sim, Não ou ArquivoTxt	Sim	"Indica se deve ser criado o Histograma do tempo de espera na fila."
10	"Hist Espera - Base"	Real	0.1	"Marca o início do histograma de tempo de espera"
11	"Hist Espera - Largura"	Real	1	"Largura de cada faixa no histograma de tempo de espera".

Shape Fonte

N	Nome do Campo	Tipo/ Valores	Default	Prompt
1	Entidade	String	Entidade1	"Entidade a ser criada a partir desta fonte"
2	"Prioridade"	Inteiro a partir de 1	1	"Prioridade da entidade criada por esta fonte sobre as outras entidades. Máxima prioridade=1. Outras: >1".
3	Atributo	Lista de Strings separada por ";"	Atributo1	"Gera atributos (opcionais) para a entidade gerada pela fonte. Obs: para criar mais de um, basta separá-los por ponto e vírgula".
4	"Histograma"	Lista de Strings separada por ";"	(Atributo1,0.5,1)	"Histogramas de atributos ao final da simulação. Indicar (Atributo, Base, Largura). Separar vários atributos por ponto e vírgula".

Shape FilaRec

N	Nome do Campo	Tipo/ Valores	Default	Prompt
1	"Inicial"	Inteiro a partir de 0	0 (zero)	"Número de entidades na fila no início da simulação."
2	"Nome Entidade"	String		"Nome da entidade associada ao recurso."
3	"Prioridade"	Inteiro a partir de 1	1	"Indica a prioridade sobre outras filas, quando necessário. Maior=1."
4	"Disciplina"	FIFO LIFO PorPrioridade PorAtributo(Nome)	FIFO	"Disciplina de entrada e saída na fila. FIFO - first in first out, LIFO - last in first out e PorAtributo - ordem por atributo"
5	"Capacidade"	Infinita ou Inteiro a partir de 1	Infinita	"Capacidade da fila. Pode ser Infinita ou ter limite (número de 1 em diante)"
6	"ExcedeCapacidade"	BloqueiaAtividade ou SaiDoSistema	Bloqueia Atividade	"Indica o que fazer com a entidade que chegar com a fila que estiver acima do limite da capacidade"
7	"Histograma Tamanho"	Sim, Não ou ArquivoTxt	Sim	"Indica se deve ser criado o Histograma do Tamanho da Fila."
8	"Hist Tam - Base"	Inteiro	0.5	"Marca o início do Histograma."
9	"Hist Tam – Largura"	Inteiro a partir de 1	1	"Largura de cada faixa do histograma de tamanho da fila."
10	"Histograma Espera"	Sim, Não ou ArquivoTxt	Sim	"Indica se deve ser criado o Histograma do tempo de espera na fila."
11	"Hist Espera - Base"	Real	0.1	"Marca o início do histograma de tempo de espera"
12	"Hist Espera – Largura"	Real	1	"Largura de cada faixa no histograma de tempo de espera."

Shape Atividade

N	Nome do Campo	Tipo/ Valores	Default	Prompt
1	“Distribuição”		0 (zero)	"Número de entidades na fila no início da simulação”.
2	"Capacidade"	Inteiro	0 (infinita)	"Número de atividades que podem ocorrer ao mesmo tempo. Se não for infinita, equivale ao número de recursos”.
3	"Prioridade"	Inteiro a partir de 1	1	"Indica a prioridade sobre outras atividades, quando necessário. Maior=1”.
4	"Histograma"	Sim, Não ou ArquivoTxt	Sim	"Indica se deve ser criado o Histograma da Duração da Atividade."
5	"Histog Base"	Inteiro	0.5	"Marca o início do Histograma”.
6	"Histog Largura"	Inteiro a partir de 1	1	"Largura de cada faixa do histograma de tamanho da fila”.
10	“Define Atributo1”	String de atribuição	Vazio	Define o valor do atributo (opcional)
11	“Define Atributo2”	String de atribuição	Vazio	Define o valor do atributo (opcional)
12	“Define Atributo3”	String de atribuição	Vazio	Define o valor do atributo (opcional)

Apêndice 4

Os códigos das rotinas de simulação

O núcleo da execução da simulação é dado pelo procedimento RodandoSimul localizado na unit Util.pas. Eis o texto da procedure:

```
procedure RodandoSimul (Dono: TComponent);
Begin
     Repeat
       ProbSim.ProximoTempo; { FASE A}
       If ProbSim.Temp <= ProbSim.FimCorr
       Then Begin
            ChamaBEventos; {FASE B}
            if not Erro then
            AtravesCEventos; {FASE C}
            End
       Else Begin
            ProbSim.IncrementaCorrida;
              if not ProbSim.Fim then ProximaCorrida(Dono);
              End;
     Until ProbSim.Fim;
     CalculaDadosEstatística;
End;
```

Detalhamento da Fase A

Quando o simulador executa a linha ProbSim.ProximoTempo verifica as atividades do tipo TAtivExec que estão na lista de atividades em execução (ListaAtvExec) e toma a atividade que tem o tempo de término mais próximo. Como a lista está sempre ordenada, basta pegar a primeira atividade: ListaAtvExec.First. A seguir avança o relógio da simulação (ProbSim.Temp) para o fim desta atividade (TAtivexec(ListaAtvExec.First).TFim). O código do método ProbSim. ProximoTempo, localizado em Compnt.pas é o seguinte:

```
If  ListaAtvExec.Count >0
     Then Temp := TAtivexec(ListaAtvExec.First).TFim
     Else Temp := FimCorr +1;
If Temp > FimCorr Then Temp:= Fimcorr + 0.000001;
```

Detalhamento da Fase B

A fase B é indicada pela *procedure* ChamaBEventos, que está localizada na *Unit* Gerenciador.pas.

Esta *procedure* tem o seguinte código:

```
While (TempoCorrente) do
   While (ProximoEventoB) do
   Case NumeroProxB of
        1..997 : Fim_De_Atividades (NumProxB);
        998: FimAquecimento;
        999: FimCorrida;
        End;
```

O programa verifica pelo método ProbSim.ProximoEventoB qual o evento B mais próximo na lista de eventos tipo B (Lista_Eventos_B); de acordo com o valor encontrado irá executar o procedimento correspondente ao fim da atividade correspondente a este evento B (Fim_De_Atividades (NumProxB)).

Detalhamento da Fase C

A fase C é indicada pela procedure AtravesCEventos, que está localizada também na *Unit* Gerenciador.pas.

Esta *procedure* é bem mais complexa que a fase B, pois testa todas as condições necessárias para o início de uma atividade. Transcrevemos a seguir a parte principal do

código desta procedure; foram omitidos alguns trechos do código que são utilizados somente em situações específicas e que tornariam a leitura pouco clara.

```
For i:=0 to ListaAtv.Count-1 do
While TAtividade(ListaAtv.Items[i].Name).Pode_Iniciar(Probsim.Temp)
do Begin
   TempoAtv:=TAtividade(ListaAtv.Items[i]).Duracao_Atv;
   ProgramaAtv(TAtividade(ListaAtv.Items[i]));
   If TAtividade(ListaAtv.Items[i]).Filas_Pre.Count>0 then
   For j:=0 to TAtividade(ListaAtv.Items[i]).Filas_Pre.Count-1
   do begin
      FilaAux:=TFila(TAtividade(ListaAtv.Items[i]).Filas_Pre.Items[j]);
      Numero_Evento_B:=TAtividade(ListaAtv.Items[i]).EventosB_FilaPre[j];
      PreparaB(Numero_Evento_B,EntidadeRecolhida[j]);
      end;
   end;
```

O programa testa todas as atividades do sistema procurando aquelas que podem iniciar a cada momento. Esta verificação é feita pelo método Pode_Iniciar onde se vê, entre outras coisas, se há pelo menos uma entidade em cada uma das filas precedentes à atividade. Caso a atividade possa iniciar, é calculada a sua duração e a é feita a programação do seu término (ProgramaAtv). Ao final, são criados vários eventos B, um para cada entidade que participa da atividade (número de entidades = número de filas pré); estes eventos B contém como informações o número do Evento e o nome da entidade correspondente.