UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIENCIA DA COMPUTAÇÃO

SIMOO: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma

por

BERNARDO COPSTEIN

Tese submetida à avaliação, como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Flávio Rech Wagner
Orientador

Porto Alegre, dezembro de 1997

CIP- CATALOGAÇÃO NA PUBLICAÇÃO

Copstein, Bernardo
SIMOO: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma / por Bernardo Copstein. - Porto Alegre: CPGCC da UFRGS, 1997.
140f.: il.

Tese (Doutorado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientador: Wagner, Flávio R.

1. Simulação por Computador. 2. Ambientes de Simulação Discreta. 3. Simulação Orientada a Objetos. 4. Paradigmas de Simulação.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira
Bibliotecária responsável pela normalização de documentos: Ida Rossi

para Ana, Amanda e Rafael

Agradecimentos

Este trabalho não seria possível sem os conselhos e sugestões do meu orientador prof. Flávio Rech Wagner que contribuiram de maneira efetiva para seu desenvolvimento. Da mesma forma não poderia esquecer a colaboração expontânea do prof. Carlos Eduardo Pereira do Departamento de Engenharia Elétrica da UFRGS que por diversas vezes atendeu minhas solicitações em detrimento de seus afazeres. Da mesma forma tenho de agradecer o auxílio do João Jornada, aluno do Bacharelado em Ciência da Computação da UFRGS, sem o qual a implementação do protótipo não teria sido concluída, os conselhos e sugestões da profa. Lúcia Lisboa, a disposição da profa. Carla Freitas e do prof. Michael Mora em revisar o texto, os palpites "low level" do Prof. Marcos Vinícius Luz e a colaboração dos meus alunos de trabalho de conclusão do Bacharelado em Informática da PUCRS. É preciso destacar também o apoio da Pontifícia Universidade Católica do Rio Grande do Sul, sempre disposta a investir no aprimoramento da formação de seus funcionários. Gostaria de agradecer ainda aos meus colegas do CPGCC e da PUCRS e a meus amigos mais próximos, em especial ao Márcio, sempre dispostos a me apoiar em todos os momentos difíceis. Por fim, gostaria de agradecer a meus pais pela dedicação e compreensão, sobretudo neste último verão e a minha esposa pelo afeto, amor e sobretudo paciência, pois estar casado com alguém que está escrevendo uma tese não deve ser muito fácil.

Sumário

Lista de figuras

Lista de tabelas

Resumo

Analisando-se a literatura de simulação discreta pode-se observar que os autores, em geral, constróem seus modelos de simulação baseados em abordagens tradicionais e aceitas tais como orientação a eventos, orientação a mensagens, orientação a filas, etc. Mais recentemente encontram-se ambientes que afirmam utilizar o chamado paradigma de simulação orientado a objetos. No entanto não existe consenso na definição de tal paradigma e diferentes interpretações podem ser encontradas.

Considerando que um modelo de simulação pertence à classe dos sistemas de software, nada mais natural do que aplicar conceitos de orientação a objetos em seu desenvolvimento. Deve ficar claro, entretanto, que existe uma grande diferença entre um paradigma de simulação, isto é, as idéias e recursos usados na construção de um modelo, e um paradigma de projeto e implementação aplicado ao desenvolvimento de sistemas de simulação. Linguagens orientadas a objetos podem ser aplicadas na implementação de sistemas de simulação que utilizam conceitos de modelagem distintos. Ainda que todos possam ser chamados de sistemas orientados a objetos, pode haver confusão quanto ao significado do termo *simulação orientada a objetos*.

Este trabalho apresenta um esquema original de classificação para sistemas de simulação quanto a sua arquitetura de software onde são considerados aspectos tais como a maneira pela qual as entidades do modelo se comunicam e a forma pela qual se descrevem os eventos que alteram seu estado, entre outros. Conceitos fundamentais são identificados de maneira a definir um modelo de referência onde diferentes paradigmas de simulação possam ser caracterizados e classificados. Especial atenção é dada ao relacionamento entre os paradigmas de simulação e a orientação a objetos, onde esta última é vista como uma estratégia de projeto e implementação. Uma nova forma de caracterizar um paradigma de simulação é proposta.

SIMOO é um "framework" para simulação discreta orientada a objetos que foi construído de maneira a poder validar os conceitos propostos. Composto por uma biblioteca de classes e de uma ferramenta de edição de modelos, a principal vantagem do uso de SIMOO em relação a outras abordagens está no fato de que SIMOO permite a seleção do paradigma mais adequado à descrição de cada entidade do modelo. Esta característica permite a criação de modelos que incorporam, simultaneamente, mais de um paradigma de simulação.

A abstração básica da biblioteca de classes de SIMOO, a partir da qual são derivadas todas as entidades de um modelo, é o elemento autônomo. Este encapsula uma "thread" própria de execução e um sistema de comunicação por mensagens não tipadas que são a base de todos os paradigmas suportados por SIMOO.

A ferramenta de edição de modelos de SIMOO é chamada de MET. MET utiliza um diagrama de classes hierárquico enriquecido com recursos adequados para a construção de modelos de simulação. Além do diagrama de classes, descreve-se também um diagrama de instâncias, onde as especificações genéricas do diagrama de classes são particularizadas. A partir da especificação dos diagramas e da descrição do comportamento das entidades, MET gera um modelo executável. Finalmente, SIMOO

preocupa-se com a separação de domínios entre a descrição do modelo propriamente dito e os aspectos de visualização de resultados e interação com o usuário. Uma categoria especial de elementos autônomos chamados de monitores é provida para permitir essa separação.

Além de apresentar o "framework" SIMOO em termos de especificação e implementação, este trabalho mostra aplicações através de situações exemplo e apresenta uma análise comparativa com outros ambientes descritos na literatura.

Palavras-chave: simulação por computador, ambientes de simulação discreta, simulação orientada a objetos, paradigmas de simulação.

TITLE: "SIMOO: OBJECT ORIENTED ENVIRONMENT FOR MULTI-PARADIGM EVENT DISCRETE SIMULATION"

Abstract

When one surveys the literature on discrete simulation, it will be noticed that, in general, authors build their simulation models using traditional approaches such as event-oriented, message-oriented, queue-oriented, etc. In more recent texts, frameworks can be found that allegedly use the so called object-oriented simulation paradigm. However, there is no generally accepted definition of such a paradigm, and various interpretations can be found.

If we consider that a simulation system is an instance of the more general class of software systems, it is straightforward to apply concepts of object orientation to develop simulation systems. Nonetheless, it is important to emphasize that there is a major difference between a simulation paradigm, i.e., the principles and resources used to build the model, and a design and implementation paradigm used to develop the simulation system. Object-oriented languages can be used to implement simulation systems that follow different paradigms. If we refer to all these systems as object-oriented systems, confusion about the exact meaning of *object-oriented simulation* may occur.

This work presents an original classification of simulation systems according to their software architectures, where different aspects are taken into account, such as the way the entities in the model communicate with each other, the way one describes events that modify the entities' state, and others. In this classification, we identify basic concepts that are used to define a reference model, with which different simulation paradigms may be characterized and classified. In particular, special attention to the relationship between simulation paradigms and object-orientation is given, the latter here being seen as a strategy to design and implement simulation systems.

SIMOO is an object-oriented framework for discrete simulation, composed by a Class Library and a Model Editing Tool that has been built in order to validate the proposed concepts. The main advantage of SIMOO with respect to other frameworks is that it allows a selection of the most adequate paradigm to describe each entity in the model. As a consequence, we are able to create models that instantiate, simultaneously, more than one simulation paradigm.

The basic element of the SIMOO class library, based on which the framework derives all the entities in the model, is the *autonomous element*. This autonomous element has its own execution thread and an untyped message-based communication system that constitute the basis of all the paradigms SIMOO supports.

The SIMOO Model Editing Tool (MET) uses a hierarchical class diagram extended with resources needed to build simulation models. Along with the classe diagram, MET allows one to describe an instance diagram that details the more generic class diagram. From the diagrams and the description of the behavior of the entities, MET generates an executable model.

The SIMOO framework also emphasizes the distinction between model description and aspects of visualization and user interaction. It provides a special category of autonomous elements, the *monitors*, that implements this separation.

Besides presenting the formal specification and the implementation of the framework, in this work several examples of how to use the SIMOO are presented, along with a comparison with other existing frameworks.

Keywords: Computer simulation, discrete event simulation environment, object oriented simulation, simulation paradigms.

1 Introdução

1.1 Motivação

O uso de simulação por computador baseia-se na idéia de que uma abordagem experimental pode ser útil no suporte à tomada de decisões [PID94]. Embora o teste de uma política ou estratégia possa ser feito de forma controlada sobre o sistema real, existem problemas de natureza prática principalmente nos casos onde o sistema é de operação perigosa ou encontra-se em fase de projeto. O uso de simulação por computador permite a construção de ambientes controlados onde diferentes estratégias podem ser testadas em busca da melhor solução.

Problemas reais ou imaginários fornecem os dados a partir do qual modelos de simulação são construídos. A busca pelo entendimento e detalhamento do problema inicial resulta no *modelo conceitual* do mesmo. O modelo de simulação, que eventualmente resulta em um programa de computador, é derivado do modelo conceitual [PID94].

Sendo assim, a construção de um modelo de simulação é um processo de simplificação e abstração no qual o modelador tenta isolar os aspectos relevantes para o problema que resulta, segundo Shannon [SHA75], em uma representação de um objeto, sistema ou idéia em uma forma diferente da entidade propriamente dita, mas que apresenta o mesmo comportamento nos aspectos relativos ao estudo. Este processo de abstração depende tanto do conhecimento sobre o problema como dos objetivos da simulação.

Modelos de simulação computacionais podem corresponder a implementações de modelos matemáticos de solução analítica ou de modelos matemáticos de solução algorítmica. Os primeiros aplicam-se para certos tipos de problemas como, por exemplo, sistemas de filas simples. Em outras situações, entretanto, as equações podem vir a ter soluções demasiado complexas ou resultar em aproximações excessivas. O uso de modelos algorítmicos apresenta soluções simplificadas para uma gama maior de aplicações.

A implementação de modelos de simulação, em geral, emprega algoritmos e estratégias tradicionais conhecidas como "visões de mundo" [SHA75] ou paradigmas de simulação, a saber: orientação a eventos, orientação a processos e orientação a atividades. Estes paradigmas determinam a maneira como se descreve os eventos que alteram o estado de um modelo.

Quando as "visões de mundo" foram criadas não se tinha conhecimento de programação orientada a objetos ou de qualquer uma das técnicas modernas de projeto ou programação na área de simulação. Por esta razão, muitos trabalhos [VAU91, MAK91, BAR96, BAL97] tem apresentado variações sobre os paradigmas tradicionais. Mais recentemente existe uma preferência por estender os paradigmas tradicionais com conceitos de orientação a objetos bem como de recursos tais como animação e paralelismo entre outros. São desenvolvidos ambientes que propõem uma nova

abordagem ou paradigma de simulação. Na maioria dos casos, porém o usuário é obrigado a adaptar todas as entidades de seu modelo ao paradigma proposto.

A motivação para este trabalho surgiu da revisão do trabalho de Cota [COT92] que propõe uma revisão no paradigma de orientação a processos. Imaginou-se então uma revisão mais ampla, envolvendo as três abordagens tradicionais à luz do paradigma de objetos. Por influência dos trabalhos de Freitas [FRE90] e Pereira [PER96], acrescentou-se os aspectos de visualização, interação com o usuário e programação concorrente ao estudo.

Freitas [FRE94], no contexto da visualização de dados científicos, propõe uma abordagem unificada para análise exploratória e simulação interativa visual. Pereira [PER96] dedica-se ao estudo da análise, projeto e implementação de sistemas de tempo real. Este trabalho propõe a construção de um ambiente de simulação que permita validar as idéias propostas por Freitas, dando continuidade ao trabalho iniciado por Lindstaedt [LIN95], ao mesmo tempo que possa constituir-se em uma ferramenta útil para o projeto de sistemas de tempo real.

Freitas propõe o ambiente VISTA, ambiente genérico para visualização de dados resultantes da execução de modelos de simulação ou de qualquer outro processo computacional. Entre as características principais propostas para VISTA pode-se destacar o alto grau de interatividade e a possibilidade de se associar, dinamicamente, representações visuais ao processo sendo executado. A principal vantagem dessa abordagem reside no fato de não ser necessário prever recursos de visualização no processo gerador dos dados. No caso específico de sistemas de simulação a idéia é que os recursos para a visualização do andamento da simulação, bem como de seus resultados, não sejam parte integrante do próprio modelo. Isso garante maior flexibilidade ao usuário final do modelo que tem liberdade para escolher os aspectos que deseja visualizar, bem como facilita a separação de domínios, evitando misturar o modelo em si com os recursos de visualização.

Pereira [PER96] discute o desenvolvimento de sistemas de tempo real, em especial aplicações na área de automação industrial. Utiliza a abordagem orientada a objetos na construção de sistemas tempo-real distribuídos. O uso de orientação a objetos justifica-se por suas características de modularização e reuso, enquanto que a distribuição e concorrência são naturais a sistemas de controle. Utiliza AO C++ [PER94], um pré-processador para a linguagem C++ [STR90] que implementa a idéia de objetos autônomos [WIL94] e incorpora características temporais (tais como ativação cíclica de métodos, entre outras) como forma de facilitar a implementação de sistemas distribuídos orientados a objetos. O uso de simulação por computador integra-se naturalmente no contexto do trabalho desenvolvido por Pereira à medida que a construção de modelos auxilia na compreensão dos problemas e permite testar diferentes hipóteses. Essas vantagens são maiores se o projeto dos modelos puder ser aproveitado em grande parte na construção do sistema real. O uso de co-simulação [LEE93], facilitado pelo uso de simulação distribuída, permite, dentro de certos limites, a substituição gradual dos elementos simulados por componentes reais. Desta forma a transição entre o modelo e o sistema real permite não apenas o reuso do projeto do modelo mas de parte de sua implementação.

Visando, então, desenvolver uma ferramenta que auxilie no projeto de sistemas de tempo real explorando os conceitos propostos por Freitas, definiu-se como objetivo deste trabalho o desenvolvimento de um ambiente de simulação discreta de propósitos gerais chamado SIMOO. Tal ambiente deve permitir a especificação orientada a objetos de modelos distribuídos de maneira que, após uma primeira etapa de testes com o modelo, sua especificação possa ser aproveitada na etapas subseqüentes do processo de desenvolvimento do sistema real. Ao mesmo tempo devem ser oferecidos recursos de visualização que atendam as características que permitam a *condução (steering)* do modelo, que segundo Marshall [MAR90], implicam no controle do usuário sobre o modelo e na visualização dinâmica dos resultados. A alteração dinâmica de diversos aspectos do modelo, tais como parâmetros internos, variáveis a serem monitoradas, forma de apresentação, comportamento das entidades e até sua criação ou eliminação devem ser permitidos de forma a configurar um ambiente de simulação interativa visual [BEL87]. Finalmente, o reconhecimento da ortogonalidade existente entre o paradigma de projeto, no caso orientação a objetos, e os paradigmas de simulação facilita a construção de modelos reutilizáveis onde entidades são descritas utilizando-se o paradigma de simulação mais adequado a cada uma.

O trabalho está organizado como segue: o capítulo 2 discute uma proposta de classificação de paradigmas de simulação orientados a objetos; o capítulo 3 apresenta uma visão geral da arquitetura do sistema SIMOO; o capítulo 4 detalha o comportamento dos elementos básicos da arquitetura de SIMOO; o capítulo 5 introduz a notação desenvolvida para a representação gráfica da estrutura estática do modelo; o capítulo 6 descreve a implementação de um protótipo do sistema SIMOO; o capítulo 7 faz uma análise comparativa entre SIMOO e sistemas de mesma natureza e, finalmente, o capítulo 8 discute as conclusões e indica a continuidade da pesquisa.

O restante deste capítulo apresenta os conceitos necessários para a compreensão do restante do texto. A seção 1.2 introduz conceitos básicos sobre modelos de simulação discreta; a seção 1.3 segue a mesma linha em relação a conceitos básicos de orientação a objetos; a seção 1.4 discute aspectos de reusabilidade de software; a seção 1.5 apresenta os conceitos de meta-classes e meta-objetos; a seção 1.6 introduz noções de simulação distribuída; a seção 1.7 resume aspectos de simulação interativa visual; a seção 1.8 apresenta os objetivos do trabalho e, por fim, a seção 1.9 suas contribuições para o estado da arte.

1.2 Simulação discreta

1.2.1 Modelos de simulação

Um modelo de simulação é composto de um conjunto de entidades que interagem entre si [SHA75]. A descrição das entidades envolve uma estrutura estática e uma estrutura dinâmica. A estrutura estática define os atributos físicos de uma entidade, enquanto que a estrutura dinâmica define como seu estado altera-se ao longo do tempo. No caso de modelos de simulação discretos o estado das entidades, e conseqüentemente do modelo, só se altera em momentos bem definidos chamados de tempos de eventos.

Eventos, ou eventos discretos, marcam mudanças no estado das entidades. A tarefa do modelador é descrever o comportamento de cada entidade como uma seqüência de reações à ocorrência de eventos. Em uma simulação discreta, o fluxo da

simulação não é contínuo, movendo-se de tempo de evento em tempo de evento em intervalos raramente regulares.

A quantidade e variedade de entidades envolvidas em um modelo de simulação, bem como a independência parcial destas, requer a definição de um protocolo de comunicação entre suas entidades. Este deve permitir tanto a troca de informações e serviços como a programação de eventos.

O tempo de simulação é o elemento que determina a seqüência de eventos de uma simulação, medindo e controlando seu progresso. À variável que contém o valor do tempo simulado chama-se de relógio de simulação.

Filas modelam a espera de clientes (ou transações) por um recurso quando a taxa de chegada destes é maior que a capacidade de atendimento [GOG95]. A forma de avanço dos elementos de uma fila pode obedecer a diferentes políticas tais como: FIFO (first in, first out), LIFO (last in, first out), com ou sem prioridades, etc.

1.2.2 Descrição do comportamento das entidades

Na seção 1.2.1, destaca-se que a tarefa do modelador é descrever o comportamento de cada entidade do modelo como uma seqüência de reações a ocorrência de eventos. Existem, porém, diferentes abordagens para a descrição desses eventos [ZEI88]. Estas abordagens são chamadas, por alguns autores, de visões de mundo [SHA75] e a cada uma está associada uma estratégia ou algoritmo de simulação.

Na visão de mundo baseada na programação de eventos utilizam-se procedimentos para descrever os eventos que ocorrem e as relações de causa e efeitos entre eles. Cada evento implica em uma mudança instantânea no estado do sistema. Um procedimento que descreve um evento, além de alterar valores de variáveis pode programar novos eventos para tempos futuros ou cancelar eventos já programados. O algoritmo de simulação associado é muito simples: todos os eventos programados são armazenados em uma lista de eventos futuros ordenados pelo tempo. O algoritmo consiste em, enquanto a lista não estiver vazia, selecionar o evento mais recente, atualizar o relógio de simulação para o tempo deste evento e ativar o procedimento correspondente. Esta é a mais eficiente das três estratégias [COT92], sendo usualmente referida como abordagem orientada a eventos.

Na visão baseada na varredura de atividades, também são definidos os eventos e suas relações de causa e efeito. Neste caso, porém, o modelo define, também, eventos de guarda os quais irão ocorrer quando uma condição for encontrada. O uso deste tipo de evento obriga o algoritmo de simulação a testar todas as condições disparadoras de eventos de contingência após a ocorrência de cada evento. Isto torna esta estratégia menos eficiente que a baseada na programação de eventos. A descrição de um modelo de simulação segundo essa visão é dita orientada a atividades.

Na visão baseada na iteração entre processos, os procedimentos não descrevem eventos, mas sim processos. Um processo, entretanto, pode ser visto como uma seqüência de eventos que descrevem todo ou parte do comportamento de uma entidade. Esta abordagem possui um enfoque bastante diferenciado das outras duas. Ao invés de se descrever um evento por procedimento, conjuntos de eventos são agrupados em

procedimentos sob a forma de um processo. Como um evento é considerado uma ação instantânea, em uma rotina que descreve um evento não são encontrados comandos que simulam a passagem do tempo. A passagem do tempo é simulada através da programação de eventos futuros. Em uma rotina que descreve um processo, por outro lado, as diferentes ações que descrevem o comportamento de uma entidade consomem tempo de simulação, implicando no uso de comandos que simulam a passagem do tempo. Sempre que um destes comandos for executado, a rotina de processo será suspensa por um determinado período. Um processo suspenso é chamado de ativo se o momento da reativação é conhecido e de ocioso se o processo for programado para ser reativado quando uma dada condição ocorrer. Um processo suspenso fatalmente será reativado e, neste caso, a execução continuará do chamado ponto de reativação. Uma mesma rotina de processo pode ter mais de um ponto de reativação. Por esta razão, a implementação de uma linguagem ou conjunto de rotinas que utilize orientação a processos é mais complexa e irá exigir o uso de co-rotinas (rotinas que admitam mais de um ponto de entrada). Esta visão de mundo é conhecida como abordagem orientada a processos.

Independente da abordagem utilizada, um elemento fundamental é o escalonador de eventos. O escalonador de eventos é o responsável por manter uma lista ordenada dos eventos futuros e garantir sua execução na ordem correta. Nos modelos orientados a eventos ou atividades a presença do escalonador de eventos é sentida mais facilmente, uma vez que os eventos são explicitamente declarados. Na orientação a processos, embora a programação dos eventos não seja explícita, ela ocorre quando se usam primitivas que simulam passagem de tempo. Existe, portanto, também neste caso a figura do escalonador de eventos.

1.3 Orientação a objetos

1.3.1 Introdução

Meyers [MEY88] destaca que a qualidade de um software não pode ser avaliada por sua primeira versão e sim por sua capacidade de manter suas qualidades externas à medida que evolui. É consenso entre diversos autores [RUM91, BOO94, MEY88, BRO90] que a abordagem modular é a chave para minimizar os problemas relacionados ao reuso e extensibilidade de software. Concordam também que, quando se considera a evolução e adaptação do sistema, a modularização deve ser baseada nos dados e não nas ações visto que mesmo evoluindo um sistema tende a trabalhar com dados da mesma natureza. Sendo assim a solução é estruturar a arquitetura dos sistemas a partir de objetos, elementos que permitem agrupar dados e operações.

1.3.2 Elementos do modelo de objetos

Segundo Booch [BOO94], o modelo de objetos utiliza 4 conceitos fundamentais - abstração, encapsulamento, modularização e hierarquia - além de 3 conceitos acessórios - tipagem, concorrência e persistência. Os 4 primeiros são ditos fundamentais porque caracterizam o modelo, ou seja, modelos sem um destes elementos não podem ser considerados orientados a objetos. Os outros três são úteis, porém não fundamentais.

Uma abstração é uma simplificação que enfatiza os aspectos mais importantes e negligencia os demais. Pode-se caracterizar o comportamento de um objeto pelos serviços que oferece e operações que pode executar sobre os demais. Cada operação ou

serviço possui uma assinatura única composta por seu conjunto de parâmetros formais e tipo de retorno. Objetos servidores executam operações quando recebem requisições ou mensagens de objetos clientes. O conjunto de operações que um cliente pode solicitar a um servidor, bem como as regras que definem como essas operações podem ser solicitadas, definem o que se chama de protocolo ou interface.

Abstração e encapsulamento são conceitos complementares. Abstração se preocupa com o comportamento observável de um objeto, enquanto que encapsulamento focaliza a implementação que suporta esse comportamento. Encapsulamento é, em geral, obtido através do que se chama de *information hiding*, que é o processo de esconder as informações de um objeto que não contribuem para o entendimento de suas características essenciais, vistas externamente. Tipicamente a estrutura e a implementação dos métodos de um objeto são restritos ao escopo interno a um objeto.

Modularização é o ato de particionar um programa em componentes individuais tanto para reduzir sua complexidade como para criar um conjunto de limites bem definidos. Módulos servem de repositórios onde se armazenam classes e objetos relacionados. Em última análise, modularização é a propriedade de um sistema de ser decomposto em um conjunto de módulos fracamente acoplados.

Um conjunto de abstrações pode formar uma hierarquia e a identificação destas hierarquias em um projeto simplifica o entendimento do problema. Os dois tipos de hierarquias mais importantes dentro de um sistema são as hierarquias de herança que descrevem relações do tipo "é um" ou "é do tipo de" e as hierarquias de agregação que descrevem relações como "contém" ou "é parte de".

Gamma et alii [GAM95] definem que um tipo identifica uma determinada interface, um objeto pode ter muitos tipos e muitos objetos podem compartilhar o mesmo tipo. Por exemplo, se um objeto for capaz de atender a todas as solicitações previstas para o tipo "servidor" então será considerado do tipo "servidor". Se o mesmo objeto for também capaz de atender a todas as solicitações previstas para o tipo "cliente", então tem-se um objeto com vários tipos.

Todo programa tem pelo menos um fluxo de execução, porém sistemas que envolvem concorrência podem ter vários fluxos ou "threads". O uso de orientação a objetos é ortogonal a concorrência. Trabalhando com objetos dispõem-se de uma unidade natural de distribuição. Cada objeto pode representar uma "thread" de controle separada configurando o que se chama de objeto ativo [WIL94].

Um objeto em software ocupa certo espaço e existe por um certo período de tempo. Persistência está relacionada a objetos cujo tempo de vida transcende o tempo de vida de um programa individual. Não apenas o estado do objeto deve persistir como também sua classe, de maneira que o estado seja re-interpretado sempre da mesma forma. Persistência é a propriedade de um objeto cuja existência transcende tempo e/ou espaço.

1.3.3 Classes

A implementação de um objeto é definida por sua classe [GAM95]. Uma classe especifica a organização e representação interna dos dados de um objeto, bem como as operações que o mesmo é capaz de executar. De maneira complementar, pode-se dizer que uma classe descreve um conjunto de objetos que compartilham uma estrutura e um comportamento comuns, ou seja, compartilham a mesma implementação. Enquanto um objeto é uma entidade concreta que desempenha um papel no sistema, uma classe captura a estrutura e o comportamento comuns a todos os objetos relacionados.

Classes, em geral, não têm sentido isoladas, e, por isso mantêm relacionamentos com as demais classes do sistema. Relacionamentos entre classes indicam uma forma de compartilhamento ou alguma forma de relacionamento semântico [BOO94]. Pode-se identificar 3 tipos de relacionamentos básicos entre classes: generalização/especialização, todo/parte e associações.

Relacionamentos de generalização/especialização ou relacionamentos de herança, definem um relacionamento entre classes onde uma classe compartilha a estrutura ou o comportamento definidos em uma ou mais classes (herança simples ou múltipla respectivamente) [BOO94]. Representam uma hierarquia de abstrações na qual uma subclasse ou classe filha herda de uma ou mais superclasses ou classes pai. Tipicamente subclasses incrementam ou redefinem a estrutura e o comportamento da superclasse. Os termos generalização e especialização relacionam-se ao fato de subclasses especializarem conceitos mais genéricos definidos em superclasses. Um conceito importante é a idéia de *polimorfismo* que permite que objetos derivados de uma mesma classe reajam de formas diferentes a uma mesma mensagem definida na interface da superclasse comum.

Relacionamentos todo/parte ou de agregação estão intimamente relacionados com as hierarquias de agregação descritas na seção 1.3.2. Cabe destacar os tipos de embutimento que as partes podem ter em relação ao todo. No caso de agregações por valor existe uma dependência física entre o agregado e o agregador, de maneira que o agregado não pode existir sem o agregador. Já no caso de agregação por referência, o relacionamento ainda denota que o agregado é parte do agregador, porém, a "parte" pode existir de maneira independente. Como exemplo de agregação por valor pode-se citar a direção de um automóvel em uma agregação que descreve um carro. A direção não tem sentido sem o carro nem o carro esta completo sem ela. Como exemplo de agregação por referência cita-se as plantas de um jardim. Uma árvore, por exemplo, pode ser transplantada sem deixar de existir, bem como da mesma forma o jardim não perde sua identidade por ter uma árvore a menos.

Se por um lado relacionamentos generalização/especialização e todo/parte possuem definições consensuais na literatura, no caso de associações existe maior variação. Neste trabalho será usada a idéia de relação de conhecimento semelhante à proposta por Brock [BRO90] e Gamma [GAM95]. Relações de conhecimento implicam apenas que um objeto conhece outro, ou seja, pode enviar mensagens para ele mas não é responsável pelo mesmo. Uma relação de conhecimento denota fraco acoplamento entre os objetos.

1.4 Reusabilidade

1.4.1 Abordagens para reusabilidade

A questão da reusabilidade pode ser abordada de diferentes pontos de vista. Considerando o tipo de tecnologia utilizada, Biggerstaff [BIG87] identifica dois grandes grupos: tecnologias de composição e tecnologias de geração.

As tecnologias de composição caracterizam-se pelos conjuntos de componentes atômicos que, no caso ideal, são reutilizados sem alterações. No caso ideal dispõe-se de componentes passivos que são operados (compostos) por um agente externo. Exemplos de tais componentes podem ser esqueletos de programas, subrotinas, funções e classes de objetos.

Tecnologias de geração não são tão simples de caracterizar porque os componentes reutilizáveis não são facilmente identificáveis como entidades concretas. Os ítens reutilizáveis são padrões inseridos em um programa. Um exemplo são os geradores de aplicações.

1.4.2 Reusabilidade no projeto orientado a objetos

Conforme Johnson e Russo [JOH91], dois tipos de projetos reutilizáveis orientados a objetos são as classes abstratas e os "frameworks".

Uma classe abstrata é uma classe que não fornece a implementação de todas as operações das quais define a interface. Deve ser redefinida em subclasses de maneira que variantes específicas do comportamento esperado possam ser implementadas. Seu papel é identificar operações comuns para qualquer aplicação do domínio ao qual se aplica; sua implementação varia para cada aplicação específica [BOO94].

Um "framework" funciona como um molde para a construção de aplicações ou subsistemas dentro do domínio de uma aplicação. Basicamente, aplicações específicas são construídas especializando-se as classes do "framework" para fornecer a implementação de alguns métodos, enquanto que a maior parte da funcionalidade da aplicação é herdada (ie, código e implementação) [BOO94].

A principal característica de um "framework" é que o mesmo deve englobar a estrutura de controle da aplicação. Quando se reutiliza componentes de uma biblioteca, o programador deve desenvolver a estrutura de controle que invoca esses componentes. Quando se utiliza um "framework" escreve-se apenas o código que será invocado pelo mesmo. A estrutura de controle da aplicação é implementada pelas classes do "framework" que invoca os métodos que devem ser implementados para cada aplicação específica.

Desta forma, Johnson e Russo [JOH91] apontam classes abstratas como uma ferramenta para o reuso de projetos de pequena escala enquanto que "frameworks" permitem o reuso de projetos de grande escala. De qualquer forma, em ambos os casos, o usuário concentra-se apenas nos aspectos relevantes ao seu problema, aproveitando a experiência de quem os projetou.

1.5 Reflexão computacional

Segundo Lisboa [LIS96], algumas linguagens orientadas a objetos utilizam meta-classes para descrever a estrutura de todas as suas classes, no sentido de que as informações necessárias para a construção de uma classe se encontram em sua meta-classe. Meta-classes são classes que descrevem classes. Se a classe descritora de uma classe pode ser instanciada, então essa instância pode realizar computações sobre seus próprios dados e oferecer serviços a seus clientes. Como seu domínio é a descrição de uma classe, meta-classes podem fornecer informações relativas aos métodos, atributos, instâncias e heranças de uma determinada classe.

Meta-objetos, por outro lado, contêm informações sobre um único objeto. Segundo Foote [FOO93], meta-objetos são objetos que definem, implementam, dão suporte ou participam de qualquer maneira da execução da aplicação ou objetos de nível base. Um meta-objeto é um objeto instanciado a partir de uma classe, descreve alguns aspectos do objeto ao qual se refere e participa do processo de execução de seu referente. Entre as informações estruturais mais comuns disponíveis em um meta-objeto pode-se encontrar: a classe de um objeto, informações sobre as classes ascendentes e descendentes, informações sobre restrições de acesso às estruturas de dados, etc.

Entre as principais vantagens do uso de meta-classes e meta-objetos pode-se destacar:

- redução da complexidade: questões como - "Como se implementa um objeto persistente" - podem ser relegadas ao meta-nível livrando o programador da aplicação de conhecer detalhes do sistema de suporte.
- Separação conceitual: o programa de nível base trata apenas com conceitos próprios da aplicação, sendo os demais considerados como pertencentes ao meta nível.
- reutilização: permite a reutilização independente das classes do programa de nível base e do meta-nível.

1.6 Simulação distribuída

1.6.1 Introdução

Computadores paralelos possuem mais de um processador sendo capazes, desta forma, de executar mais de um processo simultaneamente. Em ambientes distribuídos, por outro lado, os processadores não se encontram fisicamente na mesma máquina e sim distribuídos por diversos computadores fisicamente conectados. Não é inadmissível, entretanto, que algumas das máquinas de um ambiente distribuído sejam computadores paralelos. Por fim, um ambiente de execução concorrente é aquele onde o número de processos a executar simultaneamente é maior do que o número de processadores disponíveis (distribuídos ou não). Neste caso, os processos disputam os recursos existentes.

O modelo de programação concorrente é aquele que prevê a execução de vários processos ou "threads" simultaneamente independente do fato dos processos terem de competir ou não pelos processadores disponíveis. É um modelo genérico que se adapta bem tanto em computadores com apenas um processador, como em computadores paralelos ou ambientes distribuídos.

Fujimoto [FUJ90] define **simulação paralela discreta** ou **simulação distribuída** como aquela que se refere à execução de uma única simulação discreta em um computador paralelo. Chang e Jones [CHA94], por sua vez, definem **simulação distribuída** como a que se refere a modelos de simulação construídos como um conjunto de processos executados concorrentemente em um sistema distribuído. Para evitar confusão com a visão de mundo orientada a processos e manter coerência com o que se apresenta ao longo deste trabalho, prefere-se adotar o termo **modelo concorrente** para indicar um modelo de simulação construído utilizando-se um modelo de programação concorrente e que pode ser executado tanto em máquinas com apenas um processador como em computadores paralelos ou ambientes distribuídos. Um modelo ou programa é dito escalavel de maneira transparente se nada no texto do programa limita o número de processadores que podem ser usados para executá-lo [BAE91].

Um modelo de simulação contém diversos elementos paralelizáveis. Isso ocorre sobretudo devido ao fato das entidades que compõem o modelo, em geral, desempenharem funções de maneira independente. O grande problema consiste em poder-se determinar quais atividades exercidas pelas entidades podem ser executadas em paralelo sem prejuízo da correção da simulação, ou seja, que restrições de seqüencialidade devem ser mantidas a fim de que sejam evitados erros de causalidade [FUJ90].

Uma das estratégias adotadas para facilitar a paralelização é não admitir que duas ou mais entidades utilizem variáveis compartilhadas e impor que elas se comuniquem apenas através de eventos ou mensagens com tempo de simulação (time-stamp). Isto porém não é suficiente para evitar erros de causalidade. É necessário acrescentar uma regra que diga que "nenhuma entidade pode processar eventos em ordem decrescente de tempo". Fujimoto [FUJ90] mostra que essa regra, embora suficiente, nem sempre é necessária visto que dois eventos quaisquer podem ser independentes entre si e sua execução em qualquer ordem não leva a erros de causalidade.

Diferentes estratégias podem ser usadas para contornar o problema do erro de causalidade. Independentemente do tipo de estratégia, porém, as mesmas irão recair, inevitavelmente, em duas categorias: conservativa ou otimista.

Em uma estratégia conservativa um evento só é executado depois que se tem certeza de que sua execução não irá implicar em erros de causalidade. Estratégias otimistas, por outro lado, não exigem tais garantias, optando por verificar e desfazer situações de erro.

1.6.2 O modelo do ator

Uma solução que simplifica o projeto e implementação de modelos distribuídos é utilizar o paradigma de orientação a objetos aliado ao uso do modelo do ator descrito por Wilhelm [WIL94].

O modelo do ator usa a idéia de objeto ativo. Um objeto ativo tem uma "thread" própria de execução e se comunica através de mensagens. Sua troca de mensagens, porém, pode ser assíncrona. O objeto emissor não necessita, obrigatoriamente, aguardar

o tratamento de uma mensagens antes de prosseguir em sua execução. Da mesma forma, o objeto receptor não precisa tratar imediatamente uma mensagem recebida, podendo optar por enfileirá-la para tratamento posterior.

Usando-se o modelo do ator cada entidade será mapeada para um objeto ativo. Sendo assim pode-se representar com facilidade sistemas onde existe concorrência entre as diversas entidades (concorrência inter-objetos). Se o comportamento de um dado objeto apresentar aspectos concorrentes (concorrência intra-objeto), será necessário estender o modelo do ator, permitindo que o mesmo admita que um objeto seja composto por agregações de objetos ativos.

Utilizando-se o modelo do ator, os objetos passam a ser a unidade natural de distribuição. Devido a seu casamento natural com o paradigma de orientação a objetos, não existe a necessidade de comandos específicos para indicar paralelismo, o que torna os modelos fáceis de serem portados entre ambientes que suportam e não suportam paralelismo.

Ainda que resolva o problema da determinação dos trechos paralelizáveis, o modelo do ator não impede a ocorrência de erros de causalidade devido à falta de sincronismo entre os atores. Para solucionar este problema é necessário adotar estratégias conservativas ou otimistas como descrito anteriormente.

1.7 Simulação interativa visual

Conforme Freitas [FRE90], a partir da década de 60 a simulação de sistemas discretos, que até então só se utilizava de linguagens de programação, passou a se utilizar de técnicas simplificadas de animação. Desde então, as técnicas para visualização dos resultados de simulação vêm sendo aprimoradas passando pelos gráficos 2D e 3D até técnicas mais sofisticadas.

Surgida na década de 70 e consolidada na década de 80, **simulação interativa visual** [BEL87] não se concentra apenas na visualização dos resultados da simulação, mas também na interação do usuário com o modelo sendo simulado.

Marshall et al. [MAR90] classificam as técnicas de visualização em três categorias, conforme o grau de interação entre o usuário e o modelo durante o processo de simulação: pós-processamento (pós-processing), acompanhamento (tracking) e condução (steering).

Em **pós-processamento** as imagens são geradas depois dos dados simulados. Não existe interação com a simulação propriamente dita, apenas com os dados simulados. Embora esta abordagem apresente como vantagem a possibilidade dos dados poderem ser inspecionados e estudados repetidas vezes até o completo entendimento do processo, a geração destes pode ser demorada. A principal implicação deste fato é que erros que poderiam ser percebidos no início do processo de simulação deverão aguardar que a mesma se complete, para então serem corrigidos e o processo de simulação ser reiniciado.

Em **acompanhamento** ou monitoramento, as imagens são exibidas na medida em que transcorre o processo de simulação. O usuário, entretanto, não tem possibilidade

de interagir com o modelo de simulação salvo para interromper a simulação. A vantagem, neste caso, é que os erros podem ser percebidos durante a simulação que pode ser interrompida quando tal ocorrer.

Finalmente, em **condução** há um controle do usuário sobre o modelo computacional e os resultados vão sendo visualizados dinamicamente. Diversos aspectos do modelo podem ser alterados durante a execução da simulação: parâmetros internos, variáveis a serem monitoradas, forma de apresentação das saídas, intervalos de tempo e até a inserção ou eliminação de entidades. Esta categoria corresponde ao que se chama de "Simulação Interativa Visual".

Freitas [FRE90] destaca ainda que o desenvolvimento de modelos de simulação interativa visual deve seguir alguns princípios já aceitos. São eles:

- o usuário do modelo deve participar da elaboração das representações gráficas e da especificação da interface;
- a representação gráfica do modelo deve estar disponível mesmo antes da especificação do modelo matemático;
- a interação deve ser o mais genérica possível, a fim de não limitar as questões que o usuário possa vir a fazer sobre o modelo;
- o uso do modelo deve ser feito diretamente pelo usuário final, abolindo-se a existência do "simulacionista"[1].

Por fim, Rooks [ROO91] destaca que o usuário de qualquer sistema computacional dinâmico deve ter à sua disposição o que se chama de intervenção, ou seja, um mecanismo efetivo de interação com o modelo. Modos de interação incluem:

- **Inspeção:** o usuário deve ter acesso a todos os dados relevantes do modelo para realizar experimentos.
- **Especificação:** o usuário deverá ser capaz de especificar os parâmetros do modelo de acordo com os objetivos de sua análise.
- **Visualização**: o analista deve ser capaz de visualizar os dados do modelo de forma que se ilustre a dinâmica do mesmo e seus relacionamentos de interesse.

1.8 Objetivos

Este trabalho tem por objetivo especificar e implementar um ambiente de simulação discreta de propósitos gerais que atenda as seguintes características:

- Permitir a especificação modelos usando o paradigma de orientação a objetos através de uma notação que possa ser aproveitada em um uma etapa posterior de implementação do sistema de controle das entidades reais;
- Permitir a especificação modelos de simulação distribuídos que possam ser facilmente transportados entre ambientes distribuídos ou não;
- Disponha das características de visualização e interação definidas por Marshall [MAR90] para sistemas de simulação interativa visual.

[1] Nem sempre aquele que deseja se utilizar das vantagens da simulação por computador tem condições de fazê-lo. É comum a prática de se usar um usuário "simulacionista" cuja função é construir o modelo e, em certos casos, realizar os experimentos. Neste caso, cabe ao usuário final, ou seja, aquele que necessita da simulação, analisar os resultados e implementar a solução.

Durante a análise de trabalhos correlatos, observou-se que o uso do paradigma de orientação a objetos em ambientes de simulação muitas vezes é confundido com paradigma de simulação utilizado. Além disso, o mapeamento de entidades em objetos destaca alguns aspectos do comportamento das mesmas que costumam passar despercebidos ou diluídos quando se trabalha com os paradigmas de simulação tradicionais. Em função desses aspectos definiu-se uma classificação para os paradigmas de simulação que é baseada na ortogonalidade entre diferentes aspectos de modelagem (visões de mundo, ótica do cliente ou do servidor etc) e na livre combinação entre abordagens de modelagem para cada um destes aspectos (veja capítulo 2). Observou-se também que os ambientes descritos utilizam, em geral, apenas uma abordagem ou paradigma de simulação para descrever o conjunto das entidades do modelo. Isso implica, por vezes, em descrições de comportamento de entidades pouco naturais, visto que todas devem se enquadrar ao mesmo paradigma.

Em conseqüência, acrescentaram-se duas características ao ambiente especificado:

- suportar os paradigmas definidos na classificação proposta;
- permitir o uso do paradigma mais adequado à descrição do comportamento de cada uma das entidades de um mesmo modelo.

Embora tenham de ser oferecidas facilidades para a coleta de estatísticas e geração de números aleatórios segundo distribuições de probabilidade, pela necessidade de se delimitar o escopo do trabalho, ainda que cientes de sua importância, não serão abordados os aspectos de análise estatística de dados de entrada ou de resultados. Ainda pela mesma razão, não serão feitas considerações ou análises no que diz respeito ao desempenho.

1.9 Contribuições

Este trabalho apresenta uma série de contribuições ao estado da arte no que se refere ao desenvolvimento de ambientes de simulação de uma forma geral. Entre estas pode-se destacar:

- um esquema de classificação para paradigmas de simulação baseado na ortogonalidade entre diversos aspectos de modelagem;
- uma ferramenta gráfica que permite a especificação de modelos utilizando a classificação proposta e os conceitos de orientação a objetos;
- um modelo de ambiente de simulação de propósitos gerais onde se destacam como contribuições:
    - a possibilidade de se utilizar o paradigma mais adequado à descrição de cada uma das entidades do modelo;
    - a forma como se explora os conceitos de meta-classe e meta-objeto para obter um grau máximo de interação entre o usuário e o modelo;
    - os recursos de visualização disponibilizados de maneira a permitir uma separação de domínios entre o modelo de simulação e os recursos de visualização projetados para o mesmo.

1.10 Desenvolvimento do trabalho:

O trabalho iniciou-se por uma revisão bibliográfica contemplando todo o material disponível sobre simulação orientada a objetos, visando chegar a um modelo padronizado que servisse de referencial teórico para o desenvolvimento do trabalho. Como resultado dessa primeira etapa identificou-se a ortogonalidade entre a orientação a objetos e os paradigmas de simulação bem como os aspectos distintos que devem ser levados em conta no projeto de uma entidade de simulação. Deste último tópico resultou a proposta de paradigmas de simulação orientados a objetos.

Os resultados da primeira etapa do trabalho foram validados através da submissão de um artigo a uma conferência de prestigio internacional. Foi apresentado o trabalho intitulado "The Object Oriented Approach and the Event Simulation Paradigms" [COP96] no European Simulation Multiconference em junho de 1996. Além disso apresentou-se proposta de tese perante banca examinadora em abril do mesmo ano expondo os conceitos desenvolvidos.

A segunda etapa do trabalho constitui-se na modelagem do sistema. Para o desenvolvimento do modelo estático utilizou-se uma abordagem não muito rígida inspirada nas metodologias proposta por Rumbaught [RUM91] e Shlaer/Mellor [SHL92]. O modelo comportamental do sistema foi descrito usando-se gramáticas de grafos [EHR79]. Os aspectos mais importantes dessa descrição são detalhados no capítulo 4.

Finalmente, a última etapa do trabalho constituiu-se na implementação de um protótipo. A opção pelo ambiente Windows 95 se deu por questões de disponibilidade. O editor de modelos foi implementado em linguagem Delphi pela produtividade que oferece para o desenvolvimento de interfaces. A seleção da linguagem C++ para a implementação da biblioteca de classes para simulação levou em conta sua portabilidade, já visando o futuro transporte para um ambiente distribuído.

A validação do protótipo foi feita através da submissão de um artigo (European Simulation Simuposium/1997 [COP97]) e do acompanhamento de grupos de usuários. Esses usuários enviam periodicamente suas críticas e sugestões. Essas são avaliadas e as consideradas pertinentes são acrescidas ao sistema. Esse processo de "feedback" gerou o desenvolvimento de sucessivas melhorias no protótipo.

2 Uma proposta de classificação de paradigmas de simulação orientados a objetos

2.1 Introdução

Analisando-se a literatura de simulação discreta pode-se observar que os autores, em geral, constróem seus modelos de simulação baseados em abordagens tradicionais e aceitas, tais como orientação a mensagens [CHA94], orientação a filas [VAU91], orientação a eventos [SHA75, BAR96, ABE93], entre outras. Mais recentemente encontram-se ambientes que afirmam utilizar o chamado paradigma de simulação orientado a objetos. No entanto não existe consenso na definição de tal paradigma e diferentes interpretações podem ser encontradas.

Orientação a objetos tornou-se popular nos últimos anos como uma abordagem poderosa para o desenvolvimento de sistemas computacionais complexos. Reusabilidade e extensibilidade estão entre os benefícios prometidos quando se usa seus conceitos. Considerando que um modelo de simulação pertence à classe dos sistemas de software, nada mais natural do que aplicar esses conceitos no desenvolvimento de sistemas de simulação. Deve ficar claro, entretanto, que existe uma grande diferença entre um paradigma de simulação, isto é, as idéias e recursos usados na construção de um modelo, e um paradigma de projeto e implementação aplicado ao desenvolvimento de sistemas de simulação. Linguagens orientadas a objetos podem ser aplicadas na implementação de sistemas de simulação que utilizam conceitos de modelagem distintos. Ainda que todos possam ser chamados de sistemas orientados a objetos, pode haver confusão quanto ao significado do termo *simulação orientada a objetos*.

Neste capítulo é apresentado um esquema original de classificação para sistemas de simulação quanto a sua arquitetura de software. Conceitos fundamentais são identificados de maneira a definir um modelo de referência onde diferentes paradigmas de simulação possam ser caracterizados e classificados. Especial atenção é dada ao relacionamento entre os paradigmas de simulação e a orientação a objetos, onde esta última é vista como uma estratégia de projeto e implementação. Uma nova forma de caracterizar um paradigma de simulação é proposta.

O restante deste capítulo está organizado como segue. A seção 2.2 descreve a forma como se organizam os paradigmas de simulação discreta. A seção 2.3 discute simulação orientada a objetos. A seção 2.4 apresenta uma classificação original para paradigmas de simulação e por fim a seção 2.5, através de um exemplo concreto, faz uma análise do uso de orientação a objetos em simulação aliado à forma proposta de se encarar os paradigmas de simulação.

2.2 Paradigmas de simulação discreta

Modelos de simulação discreta são coleções de entidades que interagem entre si [SHA75]. Sua descrição compreende uma estrutura estática e uma estrutura dinâmica [COT92]. A estrutura estática define os atributos de uma entidade que caracterizam seu estado. A estrutura dinâmica define como esse estado se altera ao longo do tempo. Utilizando-se como exemplo o modelo de um elevador, as propriedades estáticas irão descrever, por exemplo, sua capacidade, velocidade, o andar em que se encontra,

quantos passageiros está transportando e a situação da porta da cabina. As propriedades dinâmicas irão descrever como o elevador reage a uma chamada de andar, em que condições as portas se fecham, se o elevador sobe ou desce, etc. A figura 2.1a apresenta a estrutura estática do elevador, descrita utilizando-se a notação de Shlaer/Mellor [SHL92], enquanto que a figura 2.1b apresenta a estrutura dinâmica segundo a mesma notação. A estrutura estática é, em geral, representada por uma coleção de objetos de dados e seus valores possíveis. A definição da estrutura dinâmica, por outro lado, é mais complexa e existem diferentes aspectos a serem considerados:

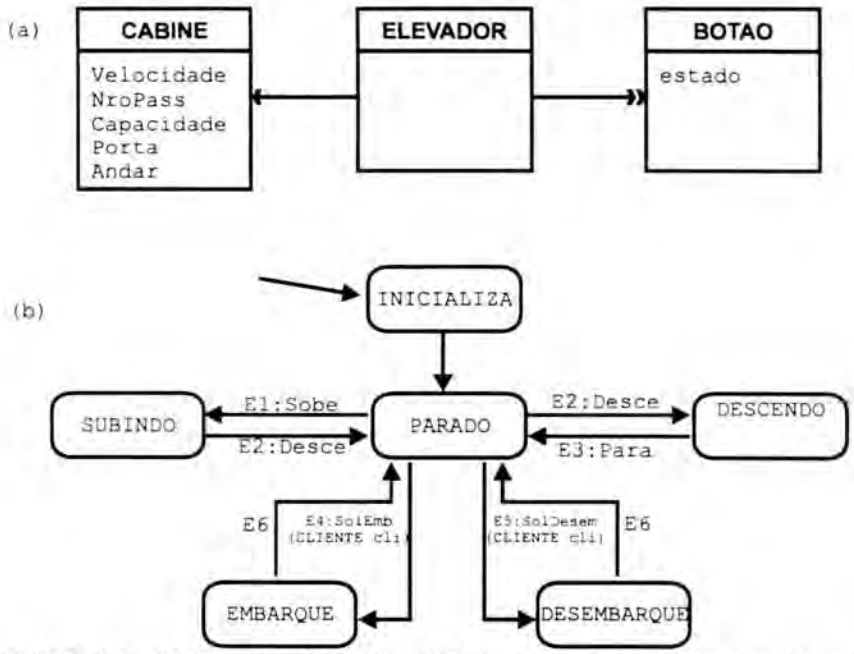


FIGURA 2.1 - Diagramas de Shlaer/Mellor que descrevem um elevador

- a maneira como se representam os eventos que definem as alterações no estado do sistema
- os recursos para comunicação entre entidades
- o ponto de vista a partir do qual o comportamento das entidades é descrito

Em modelos discretos de simulação, o comportamento de uma entidade é, em geral, definido em termos de eventos e reações a eventos. Existem diferentes abordagens para descrever a seqüência desses eventos, isto é, a maneira como são programados. São elas: orientação a eventos, processos ou atividades. Para maiores detalhes sobre estas abordagens veja a seção 1.2.2.

Entidades comunicam-se entre si para trocar informações, tarefas e programar eventos. Algumas das técnicas de comunicação mais conhecidas são o uso de portas e mensagens. Quando se usa mensagens, as entidades estão habilitadas a trocar informações com qualquer entidade cujo identificador seja conhecido. O uso de portas, por outro lado, restringe essa comunicação a canais previamente estabelecidos. O uso de portas e mensagens será discutido na seção 2.4.2.3.

Por fim, um modelo de simulação pode tanto ser descrito do ponto de vista das entidades clientes como do ponto de vista das entidades servidoras (veja seção 2.4.2.6). Cota [COT92] enfatiza que o limite entre as duas abordagens não é bem definido.

Existem diversos problemas que não podem ser modelados utilizando-se apenas um destes pontos de vista e, como conseqüência, abordagens híbridas podem ser encontradas. Algumas linguagens como GPSS [COX87] e ambientes tais SIMFACT, utilizam unicamente o ponto de vista do cliente. Outras linguagens como Simscript [CAC90] e Simula [BIR73] ou ambientes como ExSim [ZHE93] ou a biblioteca apresentada por Kocher & Lang [KOC94] são mais genéricos admitindo, inclusive, abordagens híbridas.

Estes três aspectos - comunicação, ponto de vista do cliente ou do servidor e seqüenciamento de eventos - são independentes entre si. O mecanismo de comunicação escolhido, por exemplo, não implica em nenhuma restrição sobre a escolha da abordagem para descrever o seqüenciamento dos eventos ou o ponto de vista do cliente ou do servidor. Desta forma, um modelo de simulação pode ser descrito como uma composição destes aspectos. Esta composição irá caracterizar um *paradigma de simulação*.

2.3 Simulação orientada a objetos

Em geral, um projeto de simulação progride através de seqüências lógicas e estruturadas de atividades. Para melhorar esse processo, o conceito de ambiente de simulação envolve uma linguagem de simulação e ferramentas de suporte. A linguagem de modelagem é a maior contribuição para um ambiente de simulação porque influencia diretamente a maioria dos outros recursos [TAN94].

Rumbaugh [RUM91] menciona que, quando se usa conceitos de orientação a objetos para desenvolver um sistema, o mesmo é organizado como uma coleção de objetos discretos que incorporam tanto as estruturas de dados como o comportamento. Esta definição é coerente com a idéia de que modelos de simulação podem ser vistos como uma coleção de entidades que interagem entre si. Levando-se este fato em consideração, pode-se inferir que um modelo de simulação normalmente incorpora conceitos de orientação a objetos, uma vez que este paradigma é uma forma natural de se construir modelos de simulação porque pode-se facilmente mapear entidades em objetos. Essa idéia é reforçada com o fato de que a primeira linguagem a incorporar conceitos de orientação a objetos foi a linguagem Simula67 [BIR73].

Entidades de um modelo de simulação podem ser vistas como agregações de uma estrutura estática e de uma estrutura dinâmica. Ambas são descritas utilizando-se rotinas e estruturas de dados. Mochel & Oberweis [MOC92] destacam, porém, que as abordagens clássicas de simulação (orientação a eventos, orientação a processos e orientação a atividades) geram um código onde a descrição individual de cada entidade não é facilmente identificável. Na abordagem orientada a eventos, por exemplo, a descrição do comportamento pode encontrar-se espalhada em diversas rotinas de evento. Em um modelo de tamanho considerável, a identificação de quais rotinas descrevem o comportamento de uma determinada entidade pode não ser trivial. Além disso, não existe relação visível entre a descrição do comportamento, composta de rotinas, e a descrição da estrutura estática, em geral composta por variáveis e estruturas de dados. Na abordagem orientada a processos, o problema é minimizado no que diz respeito à descrição do comportamento na medida em que uma única rotina, uma rotina de processo, descreve todo o ciclo de vida da entidade. O problema persiste, porém, no que diz respeito ao relacionamento entre a estrutura estática e a estrutura dinâmica da

entidade. Além desses aspectos, nenhuma das abordagens possui construções que permitam identificar com clareza os mecanismos de comunicação utilizados pelas entidades para troca de informações entre outros fatores. Este tipo de projeto produz, em geral, código muito rápido mas, infelizmente, difícil de ler e de manter.

A descrição individualizada das entidades de simulação é um problema de projeto de software. A solução está relacionada com a escolha de uma metodologia de projeto/implementação que enfatize a modularização e o encapsulamento. Diversos autores [JAV93, BIS91, ARO93] apontam como melhor solução o uso da abordagem orientada a objetos. A compatibilidade entre a orientação a objetos e os formalismos de simulação discreta tem sido destacada [KIM92]. O mapeamento de entidades em objetos é uma forma natural de se agregar a descrição das entidades.

Ao mapear-se entidades de simulação em objetos, entretanto, alguns aspectos tem de ser levados em consideração. De acordo com Booch [BOO94], o estado de um objeto é descrito pelo conjunto de seus atributos de dados e seus valores possíveis. Abell e Judd [ABE93], entretanto, afirmam que o estado de uma entidade de simulação é definido pelo subconjunto de seus atributos que são especificamente usados para descrever o comportamento da entidade. Por exemplo, em um objeto que representa um "buffer", o estado pode ser definido apenas pelo número de peças armazenadas na fila. No que se refere às mensagens, em orientação a objetos uma mensagem usualmente corresponde à ativação de um processo do objeto receptor. Em modelos de simulação, entretanto, a troca de mensagens é mais complexa. São necessárias mensagens temporizadas de maneira a permitir o escalonamento de eventos futuros. Mensagens não temporizadas são úteis apenas para operações de consulta aos atributos de uma entidade ou para o acionamento de operações que não implicam em avanço do tempo simulado. Sistemas de enfileiramento de mensagens bastante complexos e o uso de primitivas de seleção podem ser encontrados na literatura (veja [BAE91] como exemplo). Considerando-se os aspectos relativos à troca de mensagens, Cota [COT92] define dois tipos de entidades:

- entidades de receptor passivo, onde a entidade emissora determina o instante de tempo em que o receptor irá processar a mensagem;
- entidades de receptor ativo, onde a entidade receptora armazena as mensagens em um buffer para processamento posterior.

Considerando-se o que foi colocado, pode-se identificar claramente que um ambiente de simulação deve distinguir o paradigma de simulação da estratégia de projeto/implementação oferecida. Desta forma, pode-se definir ambientes de simulação orientados a objetos como sendo aqueles que oferecem os conceitos de orientação a objetos como uma estratégia de projeto e implementação, independentemente dos paradigmas de simulação implementados pelo ambiente.

2.4 Proposta de classificação de paradigmas de simulação

2.4.1 Introdução

As entidades são o elemento fundamental de um modelo de simulação. Pode-se dizer que um modelo de simulação nada mais é do que a descrição do comportamento de um conjunto de entidades. Desta forma, a identificação das entidades que compõem

um modelo, bem como a análise dos aspectos de seu comportamento que são relevantes para o problema em questão são, conseqüentemente, as etapas mais importantes do processo de modelagem.

No processo tradicional de modelagem, após o estudo do problema, define-se o paradigma de simulação que será usado e adequa-se todas as entidades do modelo ao paradigma escolhido. Quando as entidades são usadas como unidade de modularização (no caso deste trabalho, mapeando-as para objetos) percebe-se que é possível identificar cada entidade como um modelo individual. Trabalhando-se desta forma, não existe a necessidade de todas as entidades serem descritas segundo o mesmo paradigma de simulação. Para cada entidade pode-se adotar o paradigma mais conveniente. Essa flexibilidade facilita não apenas a descrição das entidades como também seu reuso.

Como foi visto na seção 2.3, ao modelar-se as entidades como unidades independentes e autônomas, percebe-se que os paradigmas tradicionais carecem de elementos para descrever outros aspectos que se tornam relevantes. É necessário, então, revisar e estender os paradigmas de simulação tradicionais.

Neste trabalho propõe-se uma classificação para entidades de modelos de simulação discreta baseada nos diferentes aspectos envolvidos na descrição destas. Tais aspectos são identificados e as respectivas abordagens de modelagem apresentadas. A partir de tal classificação, será assumido que o paradigma de simulação utilizado para a concepção de uma determinada entidade será nomeado a partir do conjunto das abordagens selecionadas para a descrição dos diferentes aspectos desta mesma entidade. A classificação diz respeito às entidades e não ao modelo porque, como já foi mencionado, considera-se as entidades como pequenos modelos individuais que, agrupados hierarquicamente ou não, formam modelos mais complexos. Classificam-se as entidades:

- quanto ao atendimento de solicitações;
- quanto à forma de descrição dos eventos;
- quanto ao mecanismo utilizado para troca de informações;
- quanto à autonomia da entidade;
- quanto ao tempo de permanência no sistema;
- quanto à capacidade de tomar iniciativas;
- quanto à ótica pela qual é feita a descrição do comportamento das entidades.

As seções seguintes são dedicadas a descrever cada uma destas categorias individualmente.

2.4.2 Classificação das entidades quanto ao atendimento de solicitações

Durante uma simulação, uma entidade é freqüentemente solicitada a fornecer informações ou executar ações. Dependendo do tipo de entidade, esta poderá ter a capacidade de escolher entre atender ou não a uma solicitação. *Entidades de receptor passivo* não têm poder de escolha. Neste caso a entidade reage no momento da solicitação, sendo que a entidade emissora determina, através de uma solicitação, quando a reação irá ocorrer. *Entidades de receptor ativo*, por outro lado, possuem "buffers" ou filas para armazenar as solicitações recebidas, podendo escolher o momento em que as mesmas serão atendidas ou simplesmente recusadas [COT92].

Pode-se considerar ainda uma terceira categoria: *Entidades de Receptor Semi-Ativo*. Entidades de receptor semi-ativo caracterizam-se por possuir uma fila de mensagens, ou seja, a entidade emissora não determina o momento em que a solicitação será atendida. A entidade receptora, porém não tem capacidade para alterar a ordem em que as mensagens serão atendidas, ou seja, as mensagens serão atendidas, quando houver disponibilidade, na ordem em que foram recebidas.

2.4.3 Classificação das entidades quanto à forma de descrição dos eventos

O comportamento de uma entidade em um modelo de simulação discreto é sempre descrito em termos de eventos, ou seja, das ações instantâneas que provocam alterações nos estados do modelo. Existem, porém, diferentes abordagens para a descrição desses eventos [ZEI88]. Estas abordagens, orientação a eventos, orientação a processos e orientação a atividades são chamadas de visões de mundo [COT92] e a cada uma está associada uma estratégia ou algoritmo de simulação (veja seção 1.2.2).

2.4.4 Classificação quanto ao mecanismo utilizado pelas entidades para troca de informações

Em um modelo de simulação, as entidades se comunicam de maneira a trocar informações e serviços. Diferentes técnicas podem ser usadas para efetivar tal comunicação.

O método mais simples é aquele no qual as entidades podem acessar os atributos de dados umas das outras livremente, obtendo assim as informações de que necessitam. Este método será chamado de *comunicação livre*. Quando o modelo é orientado a processos, são necessários mecanismos de sincronização entre os processos, de maneira a garantir que a informação correta está sendo obtida no momento em que está disponível.

Outras técnicas prevêem o encapsulamento das informações de cada entidade. Neste caso as entidades não têm acesso direto aos atributos de dados umas das outras. Para estas situações, duas técnicas bastante conhecidas são o uso de *portas* e de *mensagens*.

O sistema de troca de mensagens parte do princípio no qual toda entidade possui um identificador único. Uma mensagem será composta, em geral, pelo identificador da entidade emissora, identificador da entidade destino, identificador do tipo de mensagem e uma lista de parâmetros. Neste tipo de sistema, toda a troca de informações entre entidades só é possível através de mensagens. Para que o sistema funcione é preciso que se estabeleçam protocolos de comunicação entre as entidades que precisam conversar.

As mensagens podem ser *síncronas* ou *assíncronas*. Quando se usam mensagens síncronas, a entidade que envia a mensagem aguarda o tratamento da mesma por parte da entidade receptora. Utilizando-se mensagens assíncronas a entidade emissora continua seu processamento independentemente se o tratamento da mensagem enviada se efetuar ou não.

Os identificadores das mensagens podem ser únicos no contexto da classe ou únicos no contexto do modelo. No primeiro caso não existe verificação de tipo em relação aos identificadores das mensagens. A entidade receptora só irá verificar se é

capaz de tratar uma mensagem recebida no momento em que se dispuser a iniciar seu tratamento. No segundo caso a verificação pode ser feita durante a compilação do modelo ou no momento do envio da mensagem.

O sistema de portas corresponde, na verdade, a uma forma diferente de se trabalhar com mensagens. Quando se usam portas, as entidades comunicam-se através de canais de comunicação previamente estabelecidos. Cada canal deve possuir, em uma de suas extremidades uma porta de saída e na outra uma ou mais portas de entrada. Toda a informação colocada em uma porta de saída é imediatamente transmitida a todas as portas de entrada a ela conectadas. As entidades não conhecem os identificadores das entidades com que desejam se comunicar, apenas os identificadores de suas próprias portas.

O sistema de mensagens caracteriza-se por comunicações entre pares de entidades. O sistema de portas facilita a distribuição de mensagens para grupos de entidades simultaneamente. A literatura apresenta variações nos sistemas de portas e mensagens em relação ao que foi descrito, porém estas são as caraterísticas fundamentais. Dose [MAK91] e ABST [GRA93] são exemplos de sistemas que usam portas, enquanto que Modes [CHA94] é um exemplo do uso de mensagens.

Tanto as portas como as mensagens podem ser usadas com entidades de receptor ativo, semi-ativo ou passivo. No sistema de comunicação livre, porém, as entidades são de receptor passivo, obrigatoriamente, visto que as entidades não têm nenhum controle sobre o acesso a suas informações.

É importante não confundir a forma com que as entidades atendem as solicitações (receptor ativo, semi-ativo ou passivo) com o fato da troca de mensagens ser síncrona ou não. Uma mensagem síncrona pode ser enviada para uma entidade que se utiliza de um receptor ativo. Neste caso a entidade emissora corre o risco de ter de aguardar até que o destinatário resolva atender sua solicitação antes de poder prosseguir em sua execução. Sinais de controle podem ser acrescentados ao protocolo de comunicação das entidades de forma a otimizar essas situações.

2.4.5 Classificação quanto à autonomia da entidade

A classificação das entidades quanto a sua autonomia está diretamente ligada ao uso do modelo do ator. Quando se usa uma linguagem orientada a objetos convencional como C++ [STR90] ou Object Pascal [JEF96], os objetos são passivos e compartilham uma única "thread" de execução. Neste caso todas as mensagens são síncronas e as entidades possuem receptor passivo. Quando se usa o modelo do ator, os objetos possuem sua "thread" própria de execução, podem enviar mensagens de forma assíncrona e as entidades podem trabalhar com receptores ativos ou semi-ativos.

As entidades de um modelo de simulação serão consideradas autônomas na medida em que forem mapeadas para objetos. O uso de *entidades autônomas* associado com mensagens não tipadas, atribui uma certa independência à entidade, permitindo, entre outras coisas, que esta não conheça, em tempo de compilação, com quem irá se comunicar durante a execução, que instâncias de um tipo de entidade não previstas em tempo de compilação podem ser ativadas durante a execução ou, ao contrário, que entidades previstas podem ser desativadas sem a necessidade de se interromper o

processo de simulação. O uso de entidades autônomas permite a interação com as mesmas, por parte do usuário, durante a simulação, seja para interferir no seu ciclo de vida ou na forma pela qual a entidade reage às mensagens ou simplesmente para obter informações.

2.4.6 Classificação das entidades quanto ao tempo de permanência no sistema

Uma entidade pode ser *permanente* ou *temporária*. Entidades permanentes existem durante todo o tempo de simulação enquanto que entidades temporárias são criadas e destruídas durante a simulação. Se imaginarmos o modelo de um guichê de atendimento onde clientes chegam, aguardam em uma fila, são atendidos e vão embora, os "clientes" são considerados entidades temporárias pois chegam, são atendidos e desaparecem do sistema. O "guichê", por outro lado, é considerado uma entidade permanente pois existirá durante toda a simulação.

2.4.7 Classificação das entidades quanto à capacidade de tomar iniciativas

Existem entidades *ativas* e entidades *passivas*. Por entidades ativas entende-se aquelas capazes de tomar a iniciativa de solicitar um serviço ou desencadear uma seqüência de ações. Entidades passivas, por outro lado, são capazes apenas de responder a solicitações. No exemplo do guichê de atendimento, pode-se projetar "cliente" como uma entidade ativa capaz de solicitar serviços. O "guichê", neste caso, poderia não tomar iniciativas sendo apenas ocupado por "clientes" em certos momentos. Em algumas linguagens de simulação entidades passivas são chamadas de *recursos*. Esta classificação, porém, não é muito rígida. Por vezes uma entidade pode ser passiva em relação a uma dada categoria de entidades e ativa em relação a outra. Por exemplo, "guichê" é passivo em relação aos "clientes" mas poderia não ser em relação a uma entidade "funcionário".

2.4.8 Classificação quanto à ótica pela qual é feita a descrição do comportamento das entidades

Um modelo de simulação, em geral, é composto por entidades que prestam serviços (servidores) e por entidades que solicitam serviços (clientes). Independente da visão de mundo utilizada, o modelo de simulação pode ser descrito tanto do ponto de vista das entidades "clientes" como das entidades "servidores". Se for adotada a ótica do cliente as entidades "cliente" serão modeladas como entidades ativas e as "servidoras" como passivas. Se for escolhida a ótica do servidor, irá ocorrer o contrário.

Para melhor entendimento pode-se utilizar como exemplo a linha de produção de uma peça qualquer. Nesta, blocos de material são manipulados em estações de trabalho até que sejam considerados acabados. Pode-se imaginar, então, os blocos como entidades temporárias (clientes), que são atendidas por estações de trabalho modeladas como entidades permanentes (servidores).

Se o modelo for descrito utilizando-se a ótica do cliente, o mesmo será descrito do ponto de vista das peças. Cada bloco de material que entrar no sistema terá conhecimento sobre todas as operações pelas quais deve passar até que seja considerado pronto e será capaz de requisitar, sucessivamente, os serviços necessários. Se for utilizada a ótica do servidor, os blocos de material conterão apenas uma identificação do tipo de peça que deve ser produzido, e cada estação de trabalho saberá o que fazer com tal tipo de peça e para qual estação deve ser enviada depois de processada.

O limite entre as duas abordagens, entretanto, não é bem definido. Nem todos os problemas de simulação se enquadram na situação proposta. Alguns modelos poderão, inclusive, adotar uma abordagem mista onde para algumas entidades será escolhida a ótica do cliente e para outras a ótica do servidor. Uma mesma entidade pode assumir uma postura de servidor perante algumas entidades e de cliente perante outras. A escolha irá depender exclusivamente do tipo de problema.

Algumas linguagens de simulação impõem não apenas a visão de mundo mas também a ótica de descrição do comportamento das entidades. GPSS [COX87], por exemplo, usa a visão orientada a processos e a ótica do cliente.

2.5 Análise de modelos orientados a objetos

Nesta seção o objetivo é analisar o uso da programação orientada a objetos na implementação de um mesmo modelo usando paradigmas de simulação diferentes. Escolheu-se variar a forma de se descrever os eventos que alteram os estados do modelo utilizando-se orientação a eventos e orientação a processos, por serem duas abordagens clássicas de simulação que, na maioria dos casos, são a base do paradigma empregado.

A seção 2.5.1 apresenta a situação problema; a seção 2.5.2 demonstra os recursos da linguagem utilizada; as seções 2.5.3 e 2.5.4 apresentam a implementação do modelo usando os paradigmas de simulação orientado a eventos e orientado a processos e, por fim, a seção 2.5.5 faz uma análise comparativa entre as duas soluções.

2.5.1 Problema-Exemplo

O problema proposto é bastante utilizado na literatura sobretudo por sua simplicidade. Consiste na simulação de um estabelecimento especializado na lavagem de veículos com o objetivo de se analisar o tempo de espera dos clientes por atendimento. O estabelecimento possui duas máquinas para lavagem automática de veículos (fig. 2.2). À medida em que os fregueses (veículos) chegam, são direcionados para serem atendidos em uma das duas máquinas. No caso de ambas estarem livres, a escolhida será sempre a máquina 1. No caso de ambas estarem ocupadas, forma-se uma fila única. Esta deverá ser atendida na medida em que as máquinas forem sendo liberadas.

A simulação será efetuada para um certo número de veículos que é parâmetro da simulação. Os tempos entre chegadas de veículos, bem como os demais tempos e distribuições de probabilidade, foram arbitrados por não influenciarem na análise que se pretende fazer.

2.5.2 Linguagem de programação utilizada

Os exemplos foram implementados em "C++" utilizando uma biblioteca de classes imaginária voltada para a construção de modelos de simulação. Tal biblioteca dispõe de uma classe "ENTIDADE" a partir da qual devem ser derivadas as descrições de todas as categorias de entidades do modelo. Desta forma as entidades passam a dispor dos recursos necessários à construção de um modelo de simulação. Entre estes recursos pode-se destacar os métodos (rotinas) que foram utilizados na implementação dos modelos:

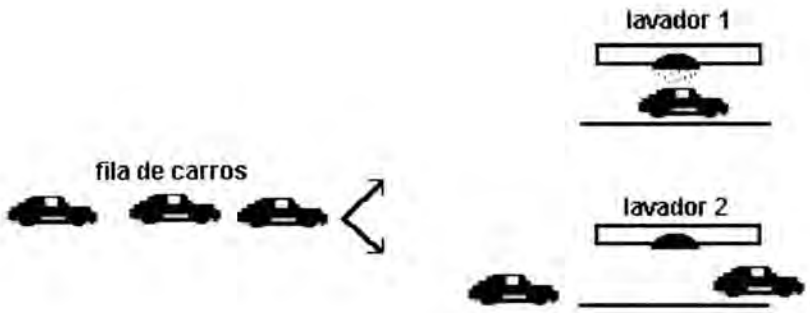


FIGURA 2.2 - Lavador de carros

- **int TempoSimulação(void)**: retorna o tempo corrente de simulação. No caso o tempo de simulação é um número inteiro.
- **void ProgramaEvento(pt_func,int tempo)**: possibilita a programação de um evento futuro. Recebe como parâmetros um ponteiro para a rotina que representa o evento programado e o "tempo do evento", ou seja, o momento no qual o evento deve ocorrer.
- **int Exponencial(float media)**: função capaz de gerar números aleatórios segundo distribuição exponencial com a média informada
- **void Aguarda(int tempo)**: suspende a execução de um processo por um certo tempo de simulação indicado no parâmetro.
- **void AguardaPor(void (*rotina)())**: suspende a execução de um processo até que outro, indicado no parâmetro, tenha sua execução concluída.
- **void ExecutaProcesso(void (*rotina)())**: dispara a execução de um processo e permite que a rotina disparadora prossiga sem aguardar a conclusão do mesmo.

Além da classe "ENTIDADE" a biblioteca fornece ainda outras classes auxiliares. Entre estas pode-se destacar a classe "QUEUE" que representa uma estrutura de dados do tipo fila com filosofia de atendimento tipo FIFO (first in, first out). Esta pode ser usada para o armazenamento de entidades temporárias.

2.5.3 Lavador de carros orientado a eventos

Esta versão do modelo da garagem utiliza um paradigma de simulação orientado a eventos descrito pela ótica do servidor com entidades de receptor passivo. A troca de informações entre as entidades segue a abordagem livre (veja seção 2.4.2.3).

O modelo foi definido a partir de 3 tipos de entidades descritos pelas seguintes classes:

- **CARRO**: classe que descreve a entidade que representa os veículos que chegam na garagem para serem lavados.
- **LAVADOR**: classe que descreve a entidade que descreve o funcionamento de uma máquina automática de lavagem de veículos.
- **LAVA_JATO**: classe que descreve a entidade que descreve a garagem propriamente dita.

O modelo é composto por uma instância da classe LAVA_JATO. A esta entidade permanente estão associadas uma fila e duas instâncias da classe LAVADOR representando as entidades que correspondem as máquinas automáticas de lavagem. A figura 2.3 apresenta o diagrama de classes do modelo segundo a metodologia de Shlaer/Mellor. LAVA_JATO gera, a intervalos pré-estabelecidos de tempo que seguem uma distribuição exponencial, eventos do tipo CHEGADA_DE_CARRO simulando a chegada dos clientes. A cada ocorrência de CHEGADA_DE_CARRO, uma entidade temporária CARRO (instância da classe CARRO) é criada e imediatamente armazenada na fila.

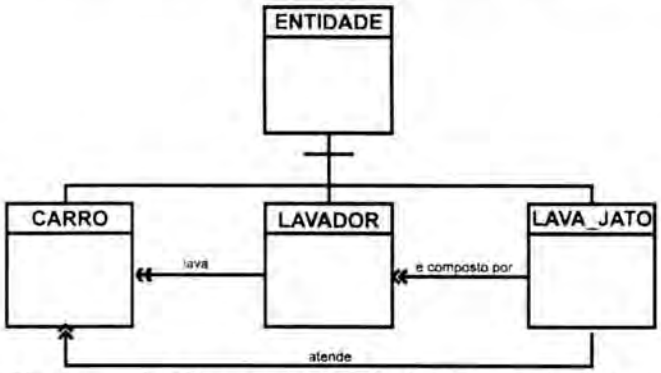


FIGURA 2.3 - Diagrama de Classes do Estabelecimento de Lavagem de Veículos

O sistema verifica se existem carros aguardando atendimento toda a vez que um veículo entra na fila e toda vez que um atendimento (lavagem) se encerra. Desta forma se evita que carros permaneçam aguardando caso exista pelo menos um lavador livre.

Quando existem as condições para um novo atendimento, um carro é retirado da fila e é gerado um evento INÍCIO_DE_LAVAGEM para o lavador selecionado. Na ocorrência deste evento, o lavador correspondente é marcado como ocupado e o tempo de lavagem é simulado através da programação de um evento FIM_DE_LAVAGEM para um tempo futuro.

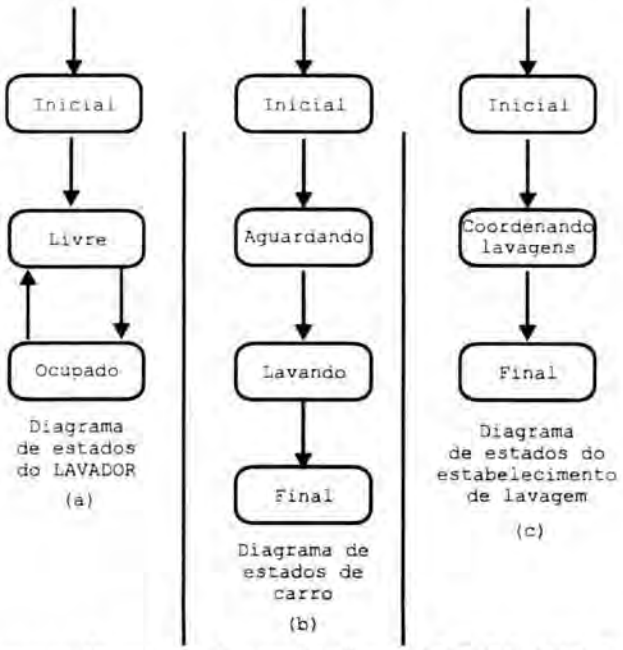


FIGURA 2.4 - Diagramas de estado das entidades CARRO,LAVADOR e LAVA_JATO

Quando ocorre um evento FIM_DE_LAVAGEM, o LAVADOR correspondente é liberado (marcado como desocupado) e a entidade temporária CARRO destruída. Procede-se, então, nova verificação na fila de clientes que aguardam, para verificar se existe mais algum a ser atendido. Esses procedimentos são repetidos até que tenha sido gerado o número de carros previstos e todos tenham sido atendidos.

O modelo é, então, composto por três tipos de entidades (descritos pelas classes CARRO, LAVADOR e LAVA_JATO) que alteram seus estados na medida em que são gerados três tipos de eventos: CHEGADA_DE_CARRO, INICIO_DE_LAVAGEM e FIM_DE_LAVAGEM. A figura 2.4 apresenta os diagramas de estados das entidades.

A implementação deste modelo em C++ foi organizada da seguinte forma: cada uma das entidades é representada por uma classe derivada da classe "ENTIDADE" como descrito no item 2.5.2. Os eventos são representados por rotinas de eventos associadas às classes que reagem aos mesmos.

É importante destacar que por ser esse modelo implementado segundo a ótica do servidor, as únicas entidades que possuem rotinas para descrição de comportamento são LAVADOR e LAVA_JATO. A entidade CARRO limita-se à descrição de seus aspectos estáticos que, no caso, compreende apenas sua identificação. Pode-se observar, também, que foram acrescentados às classes os procedimentos necessários à coleta de estatísticas.

As classes foram implementadas com o mesmo nome das entidades. As mesmas são descritas a seguir:

a)Classe CARRO (fig. 2.5): as variáveis "tempo_de_chegada_fila" e "tempo_chegada_no_lavador" são utilizadas, respectivamente, para armazenar o momento em que o carro chegou no sistema e o momento em que se iniciou o atendimento.

```
class CARRO : public ENTIDADE
  {
    int tempo_chegada_fila,
    tempo_chegada_no_lavador;
    public:
    CARRO(int t) { tempo_chegada_fila = t; }
    void InfoChegadaNoLavador(int t) { tempo_chegada_no_lavador = t; }
    int TempoEsperaFila(void)
      {
        return(tempo_chegada_no_lavador - tempo_chegada_fila);
      }
    int TempoPermSistema(int tempo_saida)
      {
        return(tempo_saida - tempo_chegada_fila);
      }
  };
```

FIGURA 2.5 - Classe "CARRO" - modelo orientado a eventos

O momento em que o carro chega no sistema é informado como parâmetro na rotina construtora da classe, enquanto que o momento em que se iniciou o atendimento é informado através da rotina "InfoChegadaNoLavador".

A classe CARRO conta ainda com as rotinas "TempoEsperaEmFila", capaz de informar quanto tempo o carro teve de aguardar por atendimento, e "TempPermSistema" que recebe o momento em que o carro deixa o sistema como parâmetro e informa quanto tempo ele permaneceu no mesmo.

b)Entidade LAVADOR (fig. 2.6): esta classe possui as variáveis "carro" e "lava_jato" que armazenam, respectivamente, referências para o carro sendo lavado e para a "garagem" que a possui. Este último tem como finalidade permitir que o LAVADOR "conheça" o LAVA_JATO podendo, assim, enviar mensagens para o mesmo.

A classe LAVADOR possui três rotinas. A rotina "Situação" é capaz de informar se o lavador está ocupado ou não. A rotina "InícioDeLavagem" é a rotina que representa o evento de mesmo nome. Quando executada providencia a ocupação do LAVADOR pelo CARRO recebido por parâmetro, bem como a programação do evento "FimDeLavagem" para um tempo futuro. Encarrega-se ainda, de informar ao CARRO o momento em que se iniciou a lavagem. Por fim, a rotina "FimDeLavagem" corresponde ao evento homônimo. Quando ativada providencia a desocupação do LAVADOR e envia uma mensagem para a "garagem" indicando o término da lavagem e devolvendo o CARRO para a mesma.

```
class LAVADOR : public ENTIDADE
  {
    CARRO *carro;
    LAVAJATO *lavajato;

    public:
    LAVADOR(LAVAJATO *lj)
      {
        lavajato = lj;
        carro = NULL;
      }
    int Situacao(void)
      {
        if (carro) return(OCUPADO);
        else return(LIVRE);
      }
    void InicioDelavagem(CARRO *car)
      {
        // Armazena o ponteiro para o carro sendo lavado
        carro = car;
        // anota o tempo de inicio do atendimento
        carro->InfoChegadaNoLavador(TempoSimulacao());
        // Programa o fim da lavagem do carro
        ProgramaEvento(FimDeLavagem(),exponencial(6,1));
     }

    void TerminoDeLavagem()
      {
        CARRO *car;
        // Libera a maquina de lavagem automatica
        car = carro;
        carro = NULL;
        // Sinaliza para o lava jato o fim da lavagem
        lavajato->FimDeLavagem(car);
      }
  };
```

FIGURA 2.6 - Classe "LAVADOR"- modelo orientado a eventos

c)Entidade LAVA_JATO (fig. 2.7): esta classe possui referências para os demais objetos que compõem a estrutura da garagem. São eles um objeto da classe QUEUE (variável "fila_carros") que suporta a fila de carros aguardando atendimento e dois objetos da classe "LAVADOR" (referenciados pelas variáveis "lav1" e "lav2"). Além disso a classe ainda possui variáveis de controle e de coleta de estatísticas. "nro_carros" e "nro_max_carros" são usadas, respectivamente, para armazenar a quantidade de carros já simulados e a quantidade de carros a ser simulada. As variáveis "tempo_medio_perm_sistema" e "tempo_medio_espera_fila" acumulam os tempos utilizados no cálculo do tempo médio que um carro aguarda na fila e tempo médio que permanece no sistema.

A classe possui ainda duas rotinas de evento e duas rotinas auxiliares.

```
class LAVA_JATO public ENTIDADE
  {
    QUEUE fila_carros;
    LAVADOR *lav1,
            *lav2;
    int nro_carros,
        nro_max_carros;
    float tempo_medio_perm_sistema,
          tempo_medio_espera_fila;

    public:
    LAVA_JATO(int n)
      {
        nro_carros = 0;
        nro_max_carros = n;
        lav1 = new LAVADOR(this);
        lav2 = new LAVADOR(this);
        tempo_medio_perm_sistema = 0;
        tempo_medio_espera_fila = 0;
      }
    ChegadaDeCarro(void);
    BuscaNaFila(void);
    FimDeLavagem(CARRO *car);
    ImpEstatisticas(void);
  };
```

FIGURA 2.7 - Classe LAVA_JATO - modelo orientado a eventos

"ChegadaDeCarro" (fig. 2.8) é a rotina que atende ao evento de mesmo nome. Providencia a criação de um novo objeto da classe CARRO e sua inserção na "fila de carros". Se o número de carros ainda não tiver sido alcançado, programa a ocorrência de um novo evento semelhante para um tempo futuro, conforme o intervalo entre chegadas estabelecido. Aciona, por fim, a rotina "BuscaNaFila" para verificar a possibilidade de iniciar um novo atendimento.

A rotina "BuscaNaFila" (fig. 2.9) é acionada tanto pela rotina "ChegadaDeCarro" como "TerminoDeLavagem". Foi implementada como rotina separada para evitar duplicação de código e melhorar o entendimento do sistema. No caso de existir um carro aguardando atendimento e um lavador livre, é responsável pela seleção de ambos e ativação do evento "InícioDeLavagem" correspondente. No caso de ambos lavadores estarem livres, a rotina é tendenciosa, escolhendo sempre o lavador 1.

```
void LAVA_JATO::ChegadaDeCarro()
  {
    CARRO *car;

    // gera um carro
    car = new CARRO(TempoSimulacao());
    // coloca o carro na fila e incrementa o nro de carros
    fila_carros.push(car);
    nro_carros++;
    // se nao gerou o nro de carros especificado
    // programa a chegada do próximo carro
    if (nro_carros < nro_max_carros)
    ProgramaEvento(ChegadaDeCarro,exponencial(5,2));
    // procura atender os carros que estão na fila
    busca_na_fila();
  }
```

FIGURA 2.8 - Rotina "ChegadaDeCarro"- modelo orientado a eventos

```
void AMBIENTE::BuscaNaFila()
  {
    LAVADOR *lav;
    CARRO *car;

    // se a fila esta vazia nao ha ninguem para atender
    if (fila_carros.Vazia()) return;
    // procura selecionar um lavador livre
    if (lav1->Situacao() == LIVRE) lav = lav1;
    else if (lav2->Situacao() == LIVRE) lav = lav2;
         else return;
    // retira um carro da fila
    car = fila_caros.pop();
    // Inicia a lavagem do carro
    lav->InicioDeLavagem(car);
  }
```

FIGURA 2.9 - Rotina "BuscaNaFila"- modelo orientado a eventos

A rotina "FimDeLavagem" (fig. 2.10) descreve a ação correspondente ao evento de mesmo nome por parte da classe LAVA_JATO. Não é acionada diretamente pela programação de um evento mas sim pela rotina homônima da classe "LAVADOR", que será acionada diretamente pela programação do evento. Sua função consiste em acumular estatísticas e eliminar o objeto da classe "CARRO" que representa o carro cuja lavagem se encerrou.

```
void LAVA_JATO::FimDeLavagem(CARRO *car)
  {
    //coleta estatisticas
    tempo_medio_perm_sistema += car->TempoPermSistema(TempoSimulacao());
    tempo_medio_espera_fila += car->TempoEsperaFila();
    // destroi o carro que saiu do sistema
    delete(car);
    // procura atender os carros que esta na fila
    busca_na_fila();
  }
```

FIGURA 2.10 - Método "FimDeLavagem"- modelo orientado a eventos

Finalmente, a rotina "ImpEstatisticas" (fig. 2.11) tem como objetivo apenas a apresentação das estatísticas resultantes da simulação. Deve ser acionada somente após o término desta.

```
void LAVA_JATO::ImpEstatisticas()
  {
    tempo_medio_perm_sistema /= nro_max_carros;
    tempo_medio_espera_fila /= nro_max_carros;
    printf("Tempo medio de permanencia no sistema: %f",
    tempo_medio_perm_sistema);
  }
```

FIGURA 2.11 - Rotina "ImpEstatisticas"- modelo orientado a eventos

A utilização efetiva de tal modelo se resume a um programa bastante simples que fica restrito a criar uma instância da entidade LAVA_JATO, disparar a rotina de criação de entidades temporárias e, após a conclusão da simulação, a rotina de coleta de estatísticas (fig. 2.12).

```
void main()
  {
    LAVA_JATO ljato(50);        // Criação do LAVA_JATO

    ljato.ChegadaDeCarro();     // Aciona o método de criação de carros
    ljato.ImpEstatisticas();    // Dispara impressão das estatísticas
  }
```

FIGURA 2.12 - Programa principal - modelo orientado a eventos

Analisando-se o projeto e a implementação do modelo, observam-se duas diferenças básicas em relação ao projeto/implementação de um modelo orientado a eventos que não utilizem recursos de orientação a objetos:

- as entidades são representadas através de objetos encapsulando todas as variáveis e rotinas necessárias à descrição de sua estrutura tanto física como comportamental. Isso facilita tanto a manutenção como a alteração e o reaproveitamento de tais entidades. Como exemplo, pode-se imaginar a situação em que seja necessário detalhar melhor o processo de lavagem propriamente dito. Basta, para tanto, acrescentar novos eventos à classe "LAVADOR", sem necessidade de alterar o restante do modelo;
- alguns eventos tem que ser representados por mais de um método de evento em classes distintas. Em cada uma, descreve-se a ação correspondente na classe associada. No modelo descrito, é o caso do evento FIM_DE_LAVAGEM que necessita ser descrito duas vezes: tanto na classe LAVADOR como em LAVA_JATO.

2.5.4 Lavador de carros orientado a processos

Esta versão do lavador utiliza um paradigma de simulação orientado a processos ao invés de eventos. Foi mantida, porém, a ótica do servidor com entidades de receptor passivo e abordagem livre para troca de mensagens de maneira a enfatizar as modificações decorrentes da alteração do paradigma de descrição do comportamento das entidades.

A primeira observação importante é que, pelo fato de ter sido alterado somente o paradigma de descrição do comportamento das entidades, apenas os aspectos relacionados à descrição do comportamento das mesmas foram alterados. Desta forma as entidades utilizadas na modelagem do problema permanecem as mesmas, sendo que

as alterações ocorrem na maneira de se descrever o fluxo dos carros através do estabelecimento de lavagem.

A entidade CARRO não sofre nenhuma alteração. Por ter sido mantida a ótica do servidor, as entidades temporárias não mudam por não contemplarem descrição de comportamento.

A lavagem de um carro passa a ser representada por um processo. Neste, o lavador é marcado como ocupado, aguarda-se um tempo que simula a lavagem propriamente dita e libera-se o lavador.

Em paralelo são executados dois processos básicos. Um processo gerador que simula a chegada dos veículos na garagem criando entidades CARRO a intervalos de tempo que seguem uma distribuição pré-definida e um processo que gerencia o atendimento dos veículos. Este monitora constantemente tanto a fila de veículos como os lavadores verificando, constantemente, se é possível disparar novo atendimento (lavagem).

A implementação deste modelo em C++ também utilizou uma classe para cada entidade. Foram criadas novamente as classes CARRO, LAVADOR e LAVA_JATO. A classe CARRO (fig. 2.13) foi inteiramente aproveitada, não sofrendo nenhuma modificação. As outras duas são descritas a seguir.

```
class CARRO
  {
    int tempo_chegada_fila,
        tempo_chegada_no_lavador;

    public:
    CARRO(int t) { tempo_chegada_fila = t; }

    void InfoChegadaNoLavador(int t) {tempo_chegada_no_lavador = t;}

    int TempoEsperaFila(void)
      {
        return(tempo_chegada_no_lavador - tempo_chegada_fila);
      }

    int TempoPermSistema(int tempo_saida)
      {
        return(tempo_saida - tempo_chegada_fila);
      }
  };
```

FIGURA 2.13 - Classe CARRO - modelo orientado a processos

a)Classe LAVADOR (fig. 2.14): a classe LAVADOR possui apenas a variável "carro" que armazena uma referência para o carro sendo lavado ou o valor nulo quando o lavador esta desocupado, assim como a variável correspondente da classe homônima do modelo orientado a eventos. Nota-se, porém, que a variável "lava_jato" foi suprimida. O fato do comportamento do lavador ser descrito agora por uma única rotina de processo torna-a desnecessária como se pode ver na rotina "Atende" (fig. 2.20).

```
class LAVADOR
  {
    CARRO *carro;

    public:
    CARRO(void)
      {
        car = NULL;
      }

    int Situacao(void)
      {
        if (carro) return(OCUPADO);
        else return(LIVRE);
      }

    void Lavagem(CARRO *car);
  };
```

FIGURA 2.14 - Descrição da entidade LAVADOR - orientado a processos

A rotina construtora desta classe apenas marca o lavador como "LIVRE" (estado inicial). A rotina "Situação" tem a mesma função que no modelo orientado a eventos, ou seja, informa se o lavador está ocupado ou não.

A rotina "Lavagem" (fig. 2.15), por fim, descreve o processo de lavagem de um veículo. Resume-se a marcar o LAVADOR como ocupado, aguardar o tempo que simula a lavagem do veículo e marcar o lavador como livre novamente. O momento de início da lavagem é informado para a entidade CARRO que representa o carro sendo lavado.

```
void LAVADOR::Lavagem(CARRO *car)
  {
    //Ocupa o lavador
    carro = car;

    // Anota tempo do inicio da lavagem
    carro->InfoChegadaNoLavador(TempoSimulacao);

    // Aguarda tempo de lavagem
    Aguarda(exponencial(6,1));

    // Libera lavador
    carro = NULL;
  }
```

FIGURA 2.15 - O Método Lavagem - orientado a processos

b)Classe LAVA_JATO (fig. 2.16): esta classe possui as mesmas variáveis que a classe homônima no modelo orientado a eventos e a funcionalidade das mesmas também se mantém. A rotina construtora também não sofreu alterações. A classe possui ainda duas rotinas que implementam os processos descritos anteriormente: "GeraCarro" e "GerenciaAtendimento".

```
class LAVA_JATO
  {
    QUEUE *fila_carros;
    LAVADOR *lav1,
            *lav2;
    int nro_max_carros,
        nro_carros;
    float tempo_medio_perm_sistema,
          tempo_medio_espera_fila;

    public:
    LAVA_JATO(int n)
      {
        nro_carros = 0;
        nro_max_carros = n;
        fila_de_carros = new QUEUE;
        lav1 = new LAVADOR;
        lav2 = new LAVADOR;
        tempo_medio_perm_sistema = 0;
        tempo_medio_espera_fila = 0;
      }
    void GeraCarros(void);
    void GerenciaAtendimento(void);
    void Atende(LAVADOR *lav,CARRO *car);
    void ImpEstatisticas(void);
  };
```

FIGURA 2.16 - Descrição da entidade LAVA_JATO - orientado a processos

A rotina "GeraCarro" (fig. 2.17), cria entidades "CARRO" na quantidade especificada para ser simulada, aguardando o tempo que simula o tempo entre chegadas entre uma geração e outra. Os CARROS criados são inseridos na fila de espera.

```
void LAVA_JATO::GeraCarros()
  {
    CARRO *car;
    int c;

    // Laço que gera a quantidade de carros especificada
    for(c=0; c<nro_max_carros; c++)
      {
        //cria um carro
        car = new CARRO(TempoSimulacao());

        // coloca o carro criado na fila
        fila_carros->Push();

        // aguarda tempo entre chegadas
        Aguarda(exponencial(5,1));
      }
  }
```

FIGURA 2.17 - Rotina "GeraCarros" - modelo orientado a processos

A rotina "GerenciaAtendimento" (fig. 2.18), monitora tanto a fila quanto os lavadores, selecionando o lavador que irá atender o próximo veículo da fila sempre que for possível e houver necessidade. Esse monitoramento se encerra quando todos os carros a serem simulados tiverem sido atendidos.

```
void LAVA_JATO::GerenciaAtendimento()
  {
    // enquanto houverem carros na fila
    while((!fila_carros->Vazia())&& (nro_carros<nro_max_carros))
      {
        if (!fila_carros->Vazia()) // Se existem carros aguardando
          if (lav1->Situacao() == LIVRE)  // Seleciona lavador
            ExecutaProcesso(Atende(lav1,fila_carros.Pop()));
           else if (lav2->Situacao() == LIVRE)
                  Atende(lav2,fila_carros.Pop());
      }
  }
```

FIGURA 2.18 - Rotina "GerenciaAtendimento"- modelo orientado a processos

A rotina "GerenciaAtendimento" se utiliza da rotina "Atende" (fig. 2.19). Esta não representa nenhum processo específico. Parte do processo de gerência do atendimento foi implementado como rotina separada apenas por uma questão de clareza.

```
void LAVA_JATO::Atende(LAVADOR *lav,CARRO *car)
  {
    // Conta mais um carro atendido
    nro_carros++;
    // aguarda o tempo de lavagem
    AguardaPor(lav->Lavagem(car));
    // acumula estatisticas
    tempo_medio_perm_sistema += car->TempoPermSistema(TempoSimulacao());
    tempo_medio_espera_fila += car->TempoEsperaFila();
    // destrói o carro que saiu do sistema
    delete(car);
  }
```

FIGURA 2.19 - Rotina "Atende" - modelo orientado a processos

A rotina "ImpEstatisticas" tem o mesmo objetivo da rotina equivalente no modelo orientado a eventos.

Finalmente, cabe observar a rotina principal (fig. 2.20). São disparados os processos de geração e controle de atendimento simultaneamente. Quando o processo de controle de atendimento se encerrar a simulação termina e, conseqüentemente, podem ser exibidos os resultados.

```
void main()
  {
    LAVA_JATO ljato(50);
    ExecutaProcesso(ljato.GeraCarros());
    AguardaPor(ljato.GerenciaAtendimento());
    ImpEstatisticas();
  }
```

FIGURA 2.20 - Programa principal - modelo orientado a processos

2.5.5 Análise comparativa

Comparando-se os modelos pode-se observar que sua estrutura estática permaneceu inalterada. Isto se explica pelo fato do paradigma de simulação utilizado ter sido alterado apenas na abordagem de descrição do comportamento das entidades. Como se pode notar, até mesmo as variáveis de classe utilizadas permaneceram as mesmas. A única variável que desapareceu no modelo orientado a processos (a variável "lavajato" da classe LAVADOR) poderia ter sido mantida se houvesse necessidade de comunicação entre o "LAVADOR" e o "LAVA_JATO". Pela maneira que se estruturou

o modelo orientado a processos tal não se fez necessário. Nota-se, porém, que a variável suprimida não descrevia um atributo do "LAVADOR" mas apenas um recurso de programação.

Uma segunda conclusão diz respeito ao fato de que o uso de orientação a objetos em modelos definidos usando-se orientação a processos, quando comparados com seu equivalente em linguagem estruturada, exige menos alterações do que se for feito o mesmo tipo de comparação com um modelo orientado a eventos. Isso ocorre porque quando se trabalha com orientação a objetos surge a necessidade de se encapsular a descrição do comportamento das entidades juntamente com sua estrutura estática. Isto leva, em modelos orientados a eventos, a que por vezes se tenha duas rotinas descrevendo diferentes aspectos de um mesmo evento associadas a entidades diferentes. Na abordagem tradicional um evento é descrito por uma única rotina, mesmo que ela necessite se referenciar a mais de uma entidade. Usando-se orientação a processos é mais natural que cada processo descreva o comportamento de uma única entidade.

Como pôde ser observado, entretanto, qualquer uma das duas abordagens é possível de se implementar utilizando-se orientação a objetos e valendo-se das vantagens desta. Reforça-se assim, o que tem-se procurado enfatizar até o momento, ou seja, o fato de que a escolha de uma abordagem orientada a objetos para o projeto ou implementação de um modelo não deve influenciar a escolha do paradigma de simulação que se deseja utilizar. As vantagens de se utilizar orientação a objetos são muitas (veja [BOO94], [RUM91] entre outros) e podem ser obtidas com qualquer paradigma de simulação.

3Visão Geral do Sistema SIMOO

3.1 Introdução

SIMOO[1] foi desenvolvido após a constatação, a partir do estudo realizado, da inexistência de um ambiente que atendesse aos objetivos estabelecidos no capítulo 1. Embora alguns trabalhos estudados apresentem características semelhantes, nenhum deles possui a abordagem desejada ou a totalidade das características pretendidas (maiores detalhes serão fornecidos posteriormente, no capítulo 7, onde uma comparação entre SIMOO e outras ferramentas de simulação é apresentada). Embora nunca se possa afirmar que uma pesquisa tenha sido exaustiva, acredita-se ter conseguido uma visão razoável do estado da arte no que diz respeito a ambientes de simulação de propósitos gerais. Como ressaltado no capítulo 1, as preocupações se concentraram nas características de modelagem e de interação desejadas, não sendo julgadas pertinentes ao escopo deste trabalho considerações relativas a desempenho, validação de modelos ou análise estatística de dados de entrada ou resultados.

Como framework [GAM95] para simulação discreta, SIMOO permite a construção de modelos de simulação discreta orientados a objetos, que incorporam recursos para permitir uma interação máxima entre o usuário e o modelo. O mapeamento de entidades de simulação em elementos autônomos, baseados na idéia de objeto ativo, oferece não apenas uma unidade de distribuição que permite a construção de modelos escalonáveis, como incentiva a construção de bibliotecas de entidades reusáveis. A abordagem hierárquica permite que o modelo seja detalhado por níveis conforme a necessidade. O uso de elementos de interface e monitores de visualização, permite ao usuário projetista manter a separação entre o domínio do modelo e os recursos de visualização e coleta de resultados. Finalmente, SIMOO suporta a concepção de paradigma de simulação proposta no capítulo 2 e permite que cada entidade do modelo seja descrita utilizando-se o paradigma de simulação que for mais adequado. Tal característica não apenas simplifica a descrição das entidades, como aumenta suas chances de reuso, visto que se elimina a possibilidade de incompatibilidade entre os paradigmas utilizados na descrição de duas entidades distintas.

O objetivo deste capítulo é apresentar uma visão geral do sistema SIMOO. Para tanto, a seção 3.2 apresenta a organização geral do framework; a seção 3.3 descreve a organização de um modelo SIMOO; a seção 3.4 relaciona as características de um elemento autônomo; a seção 3.5 descreve o mecanismo de troca de mensagens entre elementos autônomos; a seção 3.6 comenta os recursos de visualização, interação e controle de um modelo e, finalmente, a seção 3.7 comenta os paradigmas suportados.

3.2 O framework

Segundo o exposto na seção 1.4.1, SIMOO é um framework para simulação discreta que se utiliza de diferentes técnicas de reuso, permitindo ao usuário reutilizar

[1] Antes que o leitor tente adivinhar o que significa SIMOO, avisa-se que se trata de um nome e não de uma sigla. Usado no início do projeto para designar a ferramenta sendo desenvolvida como algo que lembrava "simulação orientada a objetos", terminou como designação de fato e de direito por ter caído no gosto dos usuários, sem nenhuma pretensão maior. Observam-se também pronúncias variáveis conforme o grupo de usuários. Entre estas destacam-se: SIMÓO, SIMU (como em inglês) ou SIM-Ó-Ó (preferida do autor).

diferentes aspectos de um modelo de simulação tais como interface com o usuário, componentes de visualização, componentes que representam entidades, entre outros. A arquitetura geral de SIMOO pode ser vista na figura 3.1.

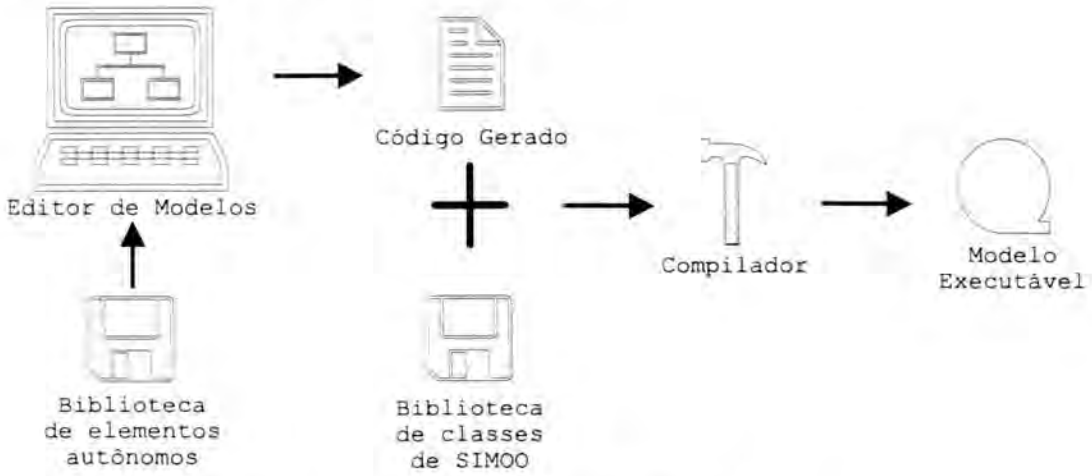


FIGURA 3.1 - Arquitetura de SIMOO

SIMOO é formado, basicamente, por uma biblioteca de classes e por uma ferramenta de geração de modelos.

A biblioteca de classes fornece os componentes necessários para a construção de um modelo de simulação. Entre o conjunto de classes disponíveis pode-se destacar as que implementam a estrutura de controle e o relógio de simulação, as que implementam a geração de números aleatórios e a coleta de estatísticas e as que servem de base para a construção das entidades do modelo e dos elementos de interação com o usuário. Estas últimas classes correspondem a classes abstratas, a partir das quais o usuário projetista do modelo deriva suas próprias entidades e elementos de visualização ou interação.

SIMOO possui uma ferramenta de edição de modelos, denominada MET - Model Editor Tool, a qual permite a construção de modelos de simulação a partir da especificação da estrutura do software que implementa o modelo. Esta abordagem, embora exija um usuário com um certo grau de conhecimento em programação, permite maior liberdade na especificação da organização do modelo ao contrário da maioria das ferramentas semelhantes (vide capítulo 7). O editor permite, ainda, a especificação gráfica da estrutura estática das entidades do modelo. No estado atual, a descrição do comportamento das entidades é feita diretamente em linguagem de programação C++, embora já se encontre em desenvolvimento um trabalho que visa substituir o uso de C++, na descrição do comportamento, por diagramas de estados ou uma gramática de grafos. Entre as características da ferramenta destaca-se a capacidade de permitir que o usuário "exporte" e "importe" descrições de entidades, facilitando a criação de bibliotecas de entidades e simplificando o reuso das mesmas. Outra característica importante é a geração de código executável. A partir da descrição do usuário, MET é capaz de gerar o código executável correspondente. Neste código são acrescentados automaticamente os recursos necessários à interação do usuário com o modelo. Entre os aspectos acrescentados pode-se destacar o painel que permite o controle do andamento da simulação, elemento básico de interação com o usuário, e a capacidade de auto-

conhecimento às entidades. Este último aspecto permite ao usuário, durante a execução da simulação, questionar e alterar diversos aspectos das entidades presentes no modelo.

3.3 Organização de um modelo

Modelos de simulação desenvolvidos com SIMOO são organizados como um conjunto de "elementos de interface" e "elementos autônomos". Elementos de interface permitem acompanhar a execução da simulação, podendo ser usados tanto para a visualização de resultados como para a entrada de dados ou alteração de parâmetros. Elementos autônomos, por outro lado, implementam as entidades do modelo. A idéia de elemento autônomo é inspirada no modelo do ator descrito por Wilhelm [WIL94]. Um elemento autônomo é um objeto ativo, ou seja, um objeto que possui "thread" própria de execução. Não possui variáveis compartilhadas ou nenhuma outra forma de se comunicar com os demais elementos autônomos ou elementos de interface senão por troca de mensagens.

SIMOO impõe uma modelagem hierárquica. No nível mais alto de abstração existe uma entidade única, o sistema sendo simulado, a qual é necessariamente refinada nos níveis inferiores constituindo uma hierarquia de agregações. Esta estratégia facilita o reuso das entidades, bem como permite que se aprofunde no grau de detalhamento à medida do necessário. Ceric [CER94] destaca as vantagens do uso de abordagens hierárquicas na construção de modelos de simulação. A figura 3.2 apresenta a estrutura hierárquica do modelo de um carro. Neste pode-se observar que a entidade "CARRO" agrega as entidades "RODA", "ASSENTO", "GUIDAO" e "MOTOR". Este último, por sua vez, é melhor detalhado agregando as entidades "CARBURADOR", "VELAS" e "PISTÕES". Em SIMOO, qualquer uma destas entidades, compostas ou não, pode ser "exportada" para posterior reuso.

Um diagrama de blocos representando a organização de um modelo SIMOO durante sua execução pode ser visto na figura 3.3. Elementos autônomos são coordenados por um gerente central chamado de "gerenciador de elementos autônomos" (GEREA). Este mantém o cadastro de todos os elementos autônomos presentes no modelo além de ser responsável pela coordenação das trocas de mensagens entre os mesmos. Este gerenciador também mantém o tempo simulado e as mensagens sincronizadas em relação a este.

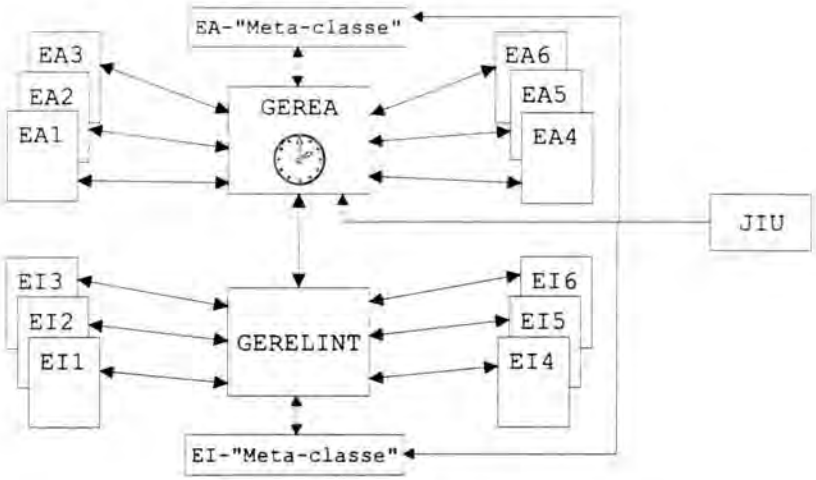
FIGURA 3.2 - Estrutura hierárquica do modelo de um carro

A fim de garantir aos usuários o acesso a todas as informações sobre o modelo durante a execução, o gerenciador mantém uma instância de meta-classe. Esta meta-

classe possui informações sobre a estrutura e serviços de todas as classes de entidades presentes no modelo. Ela conhece as tabelas do gerenciador que mantêm as listas de instâncias de entidades presentes no modelo, bem como é capaz de acionar os mecanismos de criação e remoção de entidades. A instância da meta-classe é também um elemento autônomo ainda que com privilégios especiais. Uma vez que é conhecida por todas as entidades presentes no modelo, seu uso permite que toda a solicitação dirigida ao gerenciador seja feita utilizando-se o mecanismo comum de troca de mensagens entre elementos autônomos. Além disso, garante a separação de domínios, evitando que o gerenciador tenha que conhecer as estruturas e serviços oferecidos pelas classes que descrevem as entidades.

Os elementos de interface, por outro lado, são coordenados por um gerente específico chamado de "gerenciador de elementos de interface" (GERINT). De forma semelhante ao gerenciador de elementos autônomos, o gerenciador de elementos de interface é responsável pelo cadastro dos elementos de interface presentes no modelo e pela coordenação da troca de mensagens entre os mesmos. Ao contrário dos elementos autônomos, porém, elementos de interface compartilham uma única "thread" de execução e suas mensagens são sincronizadas pelo relógio real e não pelo relógio simulado. Esta última característica se deve ao fato de que este tipo de elemento é usado para a interação com usuários reais. O gerenciador de elementos de interface também se utiliza de uma instância de meta-classe com acesso às tabelas de cadastro dos elementos de interface presentes no modelo, bem como aos seus mecanismos de criação e remoção. Na versão atual, entretanto, não se tem acesso à lista de atributos e serviços de um elemento de interface.

Devido às diferenças entre seus mecanismos de sincronização, a comunicação entre elementos autônomos e elementos de interface é intermediada por seus gerenciadores que se encarregam de contornar as diferenças.




```
GEREA = Gerenciador de Elementos Autônomos
GERELINT = Gerenciador de Elementos de Interface
EAn = elemento autônmo "n"
EIn = elemento de interface "n"
JIU = Janela de Interação com o usuário
```

FIGURA 3.3 - Organização interna de um Modelo SIMOO sendo executado

Por fim, outro elemento relevante na organização de um modelo é a janela de interação com o usuário, também representada na figura 3.3. Esta é incorporada automaticamente em qualquer modelo SIMOO e, através do relacionamento com as meta-classes, permite ao usuário criar ou remover entidades ou elementos de interface, alterar valores de parâmetros ou até mesmo a forma pela qual uma entidade responde a uma solicitação. A partir de um relacionamento especial com o gerenciador de elementos autônomos, ela permite ao usuário manipular o relógio de eventos inserindo novos eventos e alterando ou removendo os já existentes. Dessa forma o usuário pode interferir no comportamento do modelo durante a simulação.

3.4 Elementos autônomos

3.4.1 Características de um elemento autônomo

Entidades de um modelo de simulação SIMOO são mapeadas para elementos autônomos. Um elemento autônomo é um objeto que possui "thread" própria de execução e uma fila de mensagens. No momento de sua criação, um elemento autônomo recebe um estímulo (mensagem) inicial. A partir deste estímulo, o elemento autônomo pode entrar em um estado passivo, à espera de novas mensagens, ou permanecer ativo iniciando conversações com outros elementos autônomos.

Levando-se em consideração a classificação de uma entidade de simulação quanto ao atendimento de solicitações, entidades mapeadas para elementos autônomos podem ser classificadas como de receptor "semi-ativo". As mensagens que chegam para um elemento autônomo são armazenadas em uma fila e tratadas, seqüencialmente, na medida em que as que a antecederam forem sendo atendidas pela ordem. Desta forma, o momento do tratamento da mensagem não é determinado pelo emissor da mensagem, nem o tratamento da mesma pode ser antecipado ou postergado pelo receptor.

Todo o elemento autônomo possui um identificador único no contexto do modelo que o identifica dentro da hierarquia de agregações do modelo. Este identificador é formado a partir da composição de um identificador único no nível de abstração da hierarquia onde se encontra com o identificador do elemento pai na hierarquia de agregações como mostra a figura 3.4. O símbolo "@" antecedendo os identificadores indica se tratar de um elemento autônomo.

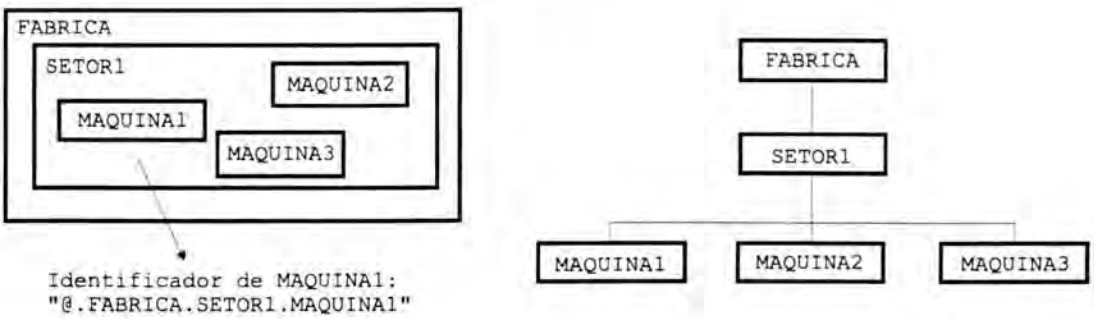


FIGURA 3.4 - Composição de um identificador de elemento autônomo

Um modelo SIMOO em execução é composto apenas por elementos autônomos atômicos. As relações de agregação e suas implicações são indicadas apenas pela forma de composição dos identificadores de elementos autônomos.

Por fim, é importante ressaltar que toda instância de elemento autônomo está associada a uma instância da classe "MetaObjeto". A classe "MetaObjeto" implementa a idéia de meta-objeto apresentada na seção 1.5. Contendo informações sobre a instância tais como classe à qual pertence e classes ancestrais na hierarquia de herança que podem ser usadas pelo próprio modelo. Além disso, ela mantém tabelas que indicam os métodos que devem ser acionados para o tratamento das mensagens previstas pela interface do elemento autônomo. Este recurso permite a alteração do comportamento do elemento autônomo durante a execução do modelo através da alteração desta tabela. A figura 3.5 apresenta um esquema da estrutura básica de um elemento autônomo.

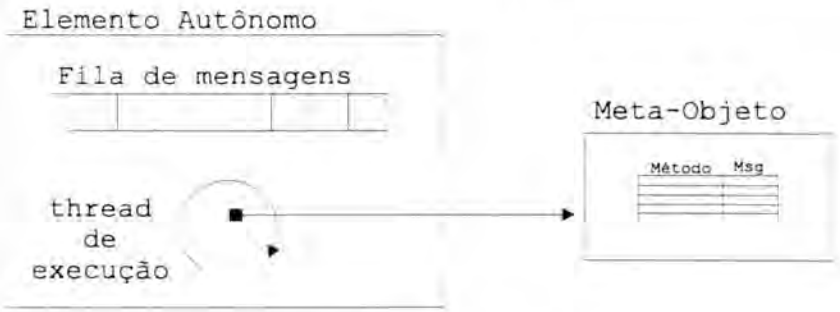


FIGURA 3.5 - Estrutura básica de um elemento autônomo

3.4.2 Mensagens de elemento autônomo

SIMOO não distingue mensagens de elementos autônomos de eventos de simulação. Uma mensagem será considerada um evento, no sentido usado em simulação, quando seu tratamento provocar uma mudança no estado da entidade.

Uma mensagem de elemento autônomo se caracteriza por conter 5 informações:

- Identificador do elemento autônomo emissor da mensagem
- Identificador do elemento autônomo destinatário da mensagem
- Identificador da mensagem propriamente dita
- Prioridade da mensagem
- Lista de argumentos

Todas as mensagens programadas por um elemento autônomo são temporizadas em relação ao relógio simulado. Um elemento autônomo pode programar mensagens para o tempo simulado corrente ou para um tempo simulado futuro. Não é admitida a programação de mensagens para tempos já decorridos.

O gerenciador de elementos autônomos mantém um escalonador de mensagens e o relógio de simulação. Toda a mensagem programada por um elemento autônomo é armazenada no escalonador de mensagens e ativada no momento oportuno.

O gerenciador se utiliza de uma estratégia conservadora para garantir que não ocorram erros de causalidade durante a execução da simulação. Esta estratégia

compreende disparar simultaneamente apenas aqueles eventos previstos para o tempo simulado corrente. Exigindo que os elementos autônomos sinalizem cada mensagem tratada e mantendo uma tabela indicativa das mensagens disparadas, o gerenciador tem condições de saber quando todas as mensagens disparadas para o tempo corrente foram tratadas. Neste momento o relógio de simulação avança e novas mensagens podem ser disparadas. Esta estratégia permite o tratamento paralelo das mensagens previstas para o mesmo tempo. Como um elemento autônomo trata suas mensagens de forma seqüencial, só existe paralelismo no tratamento das mensagens previstas para o mesmo tempo de simulação para elementos autônomos distintos. A prioridade das mensagens indica qual mensagem o gerenciador deve disparar primeiro no caso de haverem duas mensagens previstas para o mesmo elemento autônomo e para o mesmo tempo.

Esta estratégia evita a ocorrência de erros de causalidade. O projetista do modelo terá de se preocupar, porém, em evitar a ocorrência de "dead-locks" [FUJ90].

3.5 Recursos de visualização

SIMOO é rico em recursos de visualização e interação com o usuário, inclusive para controle do andamento dos experimentos. A partir de seu elemento de interface básico, é possível derivar diferentes tipos de elementos de interação que podem prover tanto a visualização de resultados como a alteração de parâmetros durante a simulação.

Todo o modelo de simulação incorpora automaticamente uma interface padrão. Construída utilizando-se elementos de interface, a interface padrão oferece um conjunto de opções que permite ao usuário acompanhar e interferir no andamento da simulação. Entre estas pode-se destacar as opções de controle do andamento da simulação (iniciar, executar passo a passo, suspender, reiniciar, etc), as opções de investigação das instâncias disponíveis, estrutura das classes e valores das variáveis, opções de inspeção e alteração do conteúdo do escalonador de eventos e opções para a criação e remoção de instâncias de elementos autônomos e elementos de interface.

Os recursos da interface padrão oferecem um conjunto razoável de opções de interação entre o usuário e o modelo. Em certos casos, porém, pode não ser admissível que o usuário do modelo deixe passar desapercebidos alguns detalhes, forçando a inserção de recursos de visualização no próprio modelo.

SIMOO disponibiliza diferentes conjuntos de elementos de interface que podem ser inseridos no modelo e novos conjuntos podem ser criados. Para que tais recursos possam detectar as mudanças de estado das entidades e atualizar seus visores é necessário que o código correspondente seja inserido no modelo. Esse código de controle de visualização pode levar à construção de um modelo confuso, onde instruções que controlam o comportamento da entidade misturam-se às necessárias para o controle de animações ou outras formas de visualização disponíveis.

Para resolver esse tipo de problema, Smalltalk utiliza o paradigma MVC [KRA88]. Mais recentemente, Gamma [GAM95] sugere o uso do padrão "Observer" para ser aplicado em qualquer uma das seguintes situações:

- quando uma abstração tem 2 aspectos, um dependente do outro, e o encapsulamento em separado pode flexibilizar o reuso individual;

- quando uma mudança em um objeto requer que outros também sejam modificados sem que se saiba quantos requerem a modificação;
- quando um objeto deve ser apto a modificar outros objetos sem assumir quem são estes outros objetos.

O MVC aplica-se especificamente à separação dos aspectos de visualização e de interação com o usuário. Utiliza um paradigma composto por três componentes (modelo, visão e controle). Os aspectos de visão são cadastrados como "dependentes" do modelo. Sempre que este sinaliza uma mudança de estado, esta sinalização é repassada a todos os "dependentes" que sabem como agir para atualizar as representações visuais.

O padrão "Observer" segue a mesma filosofia do MVC, porém, é um padrão genérico que não se limita a aspectos de visualização. Elementos a serem "observados" devem ser derivados da classe "Subject" que é capaz de cadastrar um conjunto de "observers", além disponibilizar um método para a notificação de mudanças de estado. Estas notificações são repassadas a todos os "observers" que podem tratá-las ou não.

SIMOO disponibiliza "monitores". Todo elemento autônomo possui a capacidade de cadastrar monitores. Um monitor é um elemento autônomo comum. A única restrição é que deve ser descrito utilizando-se o paradigma básico de SIMOO, ou seja, orientação a eventos e comunicação por mensagens. Quando um elemento autônomo possui monitores cadastrados, suas mensagens são duplicadas para o monitor, ou seja, toda mensagem que o elemento monitorado receber uma cópia será enviada para o elemento monitor. Pela análise dessas mensagens o monitor tem condições de tomar decisões sobre seu objetivo (por exemplo, questionar o elemento monitorado sobre seu estado para manter atualizado um elemento de visualização ou para atualizar a coleta de estatísticas). Um monitor pode controlar vários elementos autônomos assim como um elemento autônomo pode ser monitorado por vários monitores.

As principais vantagens dos monitores em relação aos "observers" ou a MVC são a inexistência de comandos para indicar a troca de estado (se tal recurso se fizer necessário pode ser simulado por uma mensagem criada especialmente para esse fim), a possibilidade de se conectar e desconectar monitores em tempo de execução e a possibilidade de monitoração seletiva: pode-se especificar que um determinado monitor deve receber cópia apenas de algumas mensagens. Essa última característica possibilita que um determinado monitor receba cópia somente daquelas mensagens que lhe dizem respeito.

No SIMOO monitores são utilizados para encapsular aspectos relativos à visualização de resultados e coleta de estatísticas.

Desta forma SIMOO oferece 3 formas de se visualizar resultados e interagir com o modelo. Pode-se criar instâncias de elementos de interface durante a execução utilizando-se a interface padrão, pode-se inserir a criação e controle destes elementos no próprio modelo ou pode-se fazer isso mantendo a independência entre os domínios utilizando-se monitores. Estas três abordagens podem ser combinadas da maneira mais adequada a cada caso.

3.6 Paradigmas suportados

Um dos objetivos de SIMOO é permitir que o usuário selecione o paradigma de simulação mais adequado à descrição do comportamento de cada entidade do modelo. No capítulo 2 é apresentada uma classificação que incorpora 7 ítens:

1. quanto ao atendimento de solicitações;
2. quanto à forma de descrição dos eventos;
3. quanto ao mecanismo utilizado para troca de informações;
4. quanto à autonomia da entidade;
5. quanto ao tempo de permanência no sistema;
6. quanto à capacidade de tomar iniciativas;
7. quanto à ótica pela qual é feita a descrição do comportamento das entidades.

SIMOO disponibiliza recursos que permitem que as entidades do modelo sejam descritas utilizando-se combinações das abordagens pertencentes à classificação. Nem todos os ítens descritos, entretanto, necessitam de suporte na biblioteca de classes de SIMOO.

Os ítens 5,6 e 7 não dependem de recursos oferecidos pela biblioteca e sim do estilo do projetista ou da finalidade da entidade sendo descrita. Qualquer elemento autônomo em SIMOO pode descrever entidades permanentes ou temporárias, passivas ou ativas e descritas segundo a ótica do cliente ou do servidor.

No que diz respeito ao item 4, SIMOO não oferece opção. Todas as entidades em SIMOO são autônomas, ou seja, possuem "thread" própria de execução e podem trocar mensagens não tipadas síncronas ou assíncronas.

Finalmente, os ítens 1,2 e 3 possuem suporte específico em SIMOO. Desta forma pode-se selecionar:

a) quanto ao atendimento de solicitações:
   - receptor passivo (atualmente não disponível);
   - receptor ativo (atualmente não disponível);
   - receptor semi-ativo;

b) quanto a forma de descrição dos eventos:
   - orientação a eventos;
   - orientação a processos;
   - orientação a atividades (atualmente não está disponível);

c) quanto ao mecanismo utilizado para a troca de informações:
   - mensagens;
   - portas;

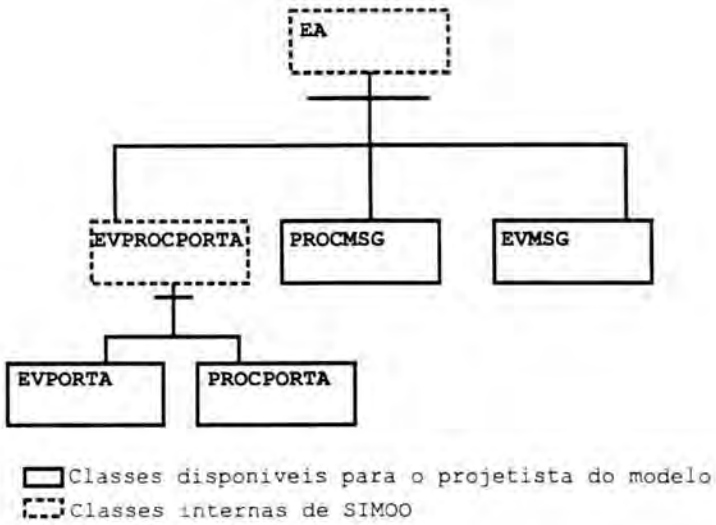


FIGURA 3.6 - Hierarquia de classes que implementam paradigmas de simulação

A figura 3.6 apresenta a hierarquia de classes que dão apoio as abordagens válidas atualmente. A idéia do elemento autônomo básico implementa os recursos que suportam a autonomia das entidades e a troca de mensagens temporizada em relação ao relógio simulado, síncronas ou assíncronas. Suporta também o conceito de co-rotina. A classe EA implementa esses recursos e a partir desta, por herança, são derivadas todas as combinações de abordagens que resultam em paradigmas válidos na versão atual.

A classe EVMSG implementa o paradigma que suporta orientação a eventos e comunicação por mensagens (na versão atual todas utilizam receptor semi-ativo). Esta classe estende a classe EA tornando visíveis apenas os métodos que permitem comunicações síncronas não instantâneas. Por comunicação síncrona instantânea entende-se aquela que permite a entidade que disparou a comunicação continuar seu processamento ainda durante o mesmo tempo de simulação. Este tipo de restrição se faz necessária para garantir a semântica do paradigma de orientação a eventos onde as rotinas de tratamento de eventos são instantâneas com relação ao tempo simulado.

A classe PROCMSG implementa o paradigma que utiliza os modelos de orientação a processos e comunicação por mensagens. A classe PROCMSG, a partir dos recursos da classe EA que implementam comunicação síncrona, implementa métodos que permitem a uma entidade aguardar a chegada de uma mensagem ou esperar a passagem de um certo tempo, disponibilizando assim os recursos necessários para o paradigma de orientação a processos. Note que tanto EVMSG como PROCMSG não precisam implementar recursos adicionais para a comunicação entre as entidades porque troca de mensagens é o paradigma básico de SIMOO.

A classe EVPROCPORTA acrescenta à classe EA os recursos necessários para a comunicação por portas. Entre estes recursos encontram-se a capacidade de armazenar listas de portas de entrada ou de saída, bem como as listas de conexões entre portas. Esta classe também é responsável por impedir a visibilidade dos recursos para a comunicação por mensagens por parte de suas derivadas.

Finalmente, as classes EVPORTA e PROCPORTA são semelhantes às classes EVMSG e PROCMSG, porém utilizando comunicação por portas ao invés de mensagens.

4 Especificação do Comportamento de um Elemento Autônomo

4.1 Introdução

O capítulo 3 apresentou uma visão geral das características e potencialidades de SIMOO. Este capítulo dedica-se a especificar em detalhes o comportamento dos elementos autônomos: a maneira como são criados e destruídos, como funciona o sistema de troca e sincronismo de mensagens, como se comunicam com elementos de interface e quais são os paradigmas de simulação suportados. Para que o entendimento seja possível esta especificação compreende o funcionamento do gerenciador de elementos autônomos e, sem muitos detalhes, do gerenciador de elementos de interface.

De maneira a facilitar a organização do capítulo, dividiu-se o comportamento de um elemento autônomo em camadas (fig 4.1). Na camada inicial (seção 4.2), chamada de elemento autônomo básico são descritos os aspectos relacionados à criação e destruição de elementos autônomos bem como seu relacionamento com o gerenciador e as possibilidades de interação com seus semelhantes. No segundo nível (seção 4.3) introduzem-se as interações com os elementos de interface e, finalmente, no terceiro nível (seção 4.4) apresenta-se o suporte aos paradigmas de simulação disponíveis até o momento.

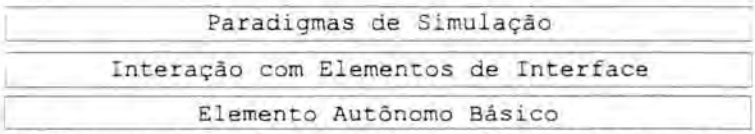


FIGURA 4.1 - Organização de um modelo SIMOO em níveis

4.2 Especificação formal do nível básico de controle de elementos autônomos

O nível básico de controle de elementos autônomos é o mais importante desta especificação porque descreve os recursos básicos de um elemento autônomo. A partir dos recursos descritos são derivados todos os paradigmas de simulação disponíveis e os que porventura venham a ser implementados. Por esta razão, as regras que conduzem o comportamento básico de um elemento autônomo são especificadas formalmente com auxílio de uma gramática de grafos [EHR79].

Gramáticas de grafos generalizam gramáticas de Chomsky de strings para grafos. Ao contrário de uma regra de Chomsky, porém, uma regra **r : L→R** não é formada apenas pelos grafos L (lado esquerdo) e R (lado direito), mas adiciona um elemento: um (homo)morfismo r mapeando arestas e vértices de L nas arestas e vértices de R.

Gramáticas de grafos especificam um sistema em termos de estados e alterações de estados. A interpretação operacional da regra **r : L→R** provê a base para esta abordagem de especificação:

- ítens em L que não tem imagem em R são deletados;
- ítens em L que tem imagem em R são preservados;
- ítens em R que não tem uma pré-imagem em L são criados.

Ao invés de se usar grafos planos compostos apenas por vértices e arestas, neste trabalho usam-se grafos tipados e com grafos com atributos, tornando a especificação mais natural, compacta e fácil de consultar.

Atributos (algebricamente especificados) são álgebras (conjunto de sinais e operações) que podem ser usados para atribuir valores aos vértices. São usados como tipos de dados básicos tais como números naturais ou strings, que não podem ser representados graficamente. Como exemplo, considere a lista de atributos de um sistema telefônico[1] (PBX) apresentado na tabela 4.1. Considere os atributos Bool e Nat. Bool denota a álgebra de valores booleanos que podem ser denotados por On e Off ou T (true) or F (false) respectivamente (mais as operações). Nat denota a álgebra dos números naturais 0,1,... incluindo as operações padrão +,-,... .

TABELA 4.1 - Atributos de um sistema de PBX

Atributo	Valor
Bool	On(T)|Off(F)
Status	Mute|Free|Busy|Call|Carrier|Ring|Wrong|Speak
Digit	0|1|2|3|4|5|6|7|8|9
Nat	números naturais
List	listas de números naturais
UserId	{user1,user2, ... ,usern,adm}

Um grafo de tipos é um grafo onde cada vértice e aresta representam algum tipo distinto de vértice/aresta em uma especificação. Cada grafo de um sistema deve ter uma interpretação em termos de um grafo de tipos. O conceito de tipagem impõe restrições estruturais nos grafos que representam os estados do sistema. Como exemplo, observe o grafo de tipos da figura 4.2 que usa os atributos da tabela 4.1. Neste pode-se distingüir os seguintes tipos de vértices: PHONE, CENTRAL, e ENVIROMENT, P:Digit, P:Sign, C:Digit e E:Act. De forma análoga pode-se observar diferentes tipos de arestas. Nota-se que não existem arestas conectando uma mensagem E-Act a CENTRAL no grafo de tipos. Isso significa que não existe nenhum estado no sistema onde uma mensagem dessas possa ser enviada à CENTRAL. Este estado é considerado inconsistente.

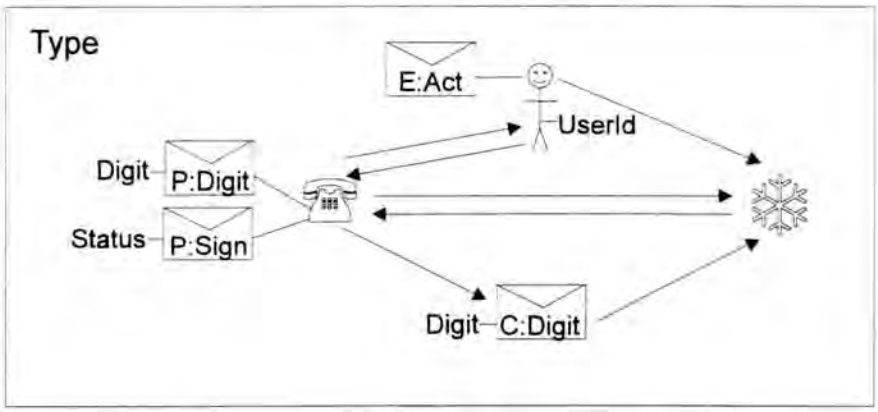


FIGURA 4.2 - Grafo de tipos do PBX

[1] Este exemplo foi extraído de "Graph Grammars for the Specification of Concurrent Systems" [RIB96].

Uma gramática de grafos é composta pelos seguintes elementos: atributos, grafo de tipos, grafo inicial e regras. A figura 4.3 apresenta uma gramática de grafos que se utiliza dos atributos da tabela 4.1, do grafo de tipos da figura 4.2, de um estado inicial (grafo) Ini e de três regras (r1, r2 e r3).

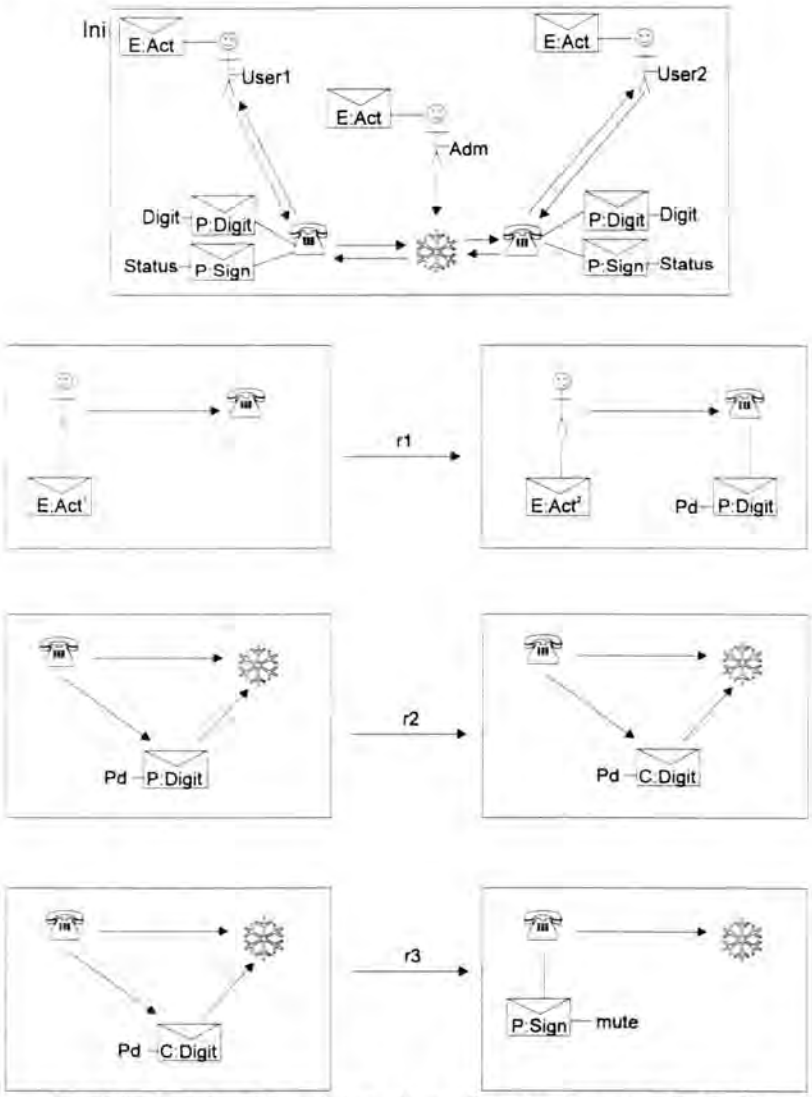


FIGURA 4.3 - Gramática de grafos do sistema de PBX

O grafo Ini mostra dois telefones conectados a uma central, onde cada telefone tem um usuário e a central tem um administrador. Os usuários e o administrador estão habilitados a agir (as ações são modeladas pelas mensagens E-Act conectadas aos mesmos). A regra r1 modela a discagem de um dígito Pd em um telefone por um usuário. O usuário e o telefone são preservados, a mensagem E-Act do usuário é deletada, outra mensagem E-Act é enviada para o usuário (criada) e uma mensagem contendo o dígito discado é enviada para o telefone (criada). A regra r2 modela o envio do dígito Pd para a central e a regra r3 o processamento desta mensagem pela central: ela envia uma mensagem para o telefone ordenando que o sinal de "linha" seja suspenso. Maiores detalhes sobre gramáticas de grafos podem ser encontrados em [LÖW93, KOR96, RIB96, EHK96].

As seções seguintes descrevem os tipos e operações utilizados na especificação do nivel básico de SIMOO, bem como as regras propriamente ditas.

4.2.1 Tipos, convenções e estado inicial

Neste nível são definidos 3 tipos compostos: GEREA que representa o gerenciador de elementos autônomos, EA que representa um elemento autônomo e MSGCTRL usado nas comunicações (mensagens) entre os mesmos e seu gerenciador. MSGCTRL representa mensagens de controle e não as mensagens que circulam entre as entidades durante uma simulação. A tabela 4.2 apresenta os tipos básicos utilizados na composição dos tipos compostos, enquanto que a tabela 4.3 apresenta os tipos compostos. Os tipos EA e MSGCTRL possuem atributos compostos do tipo MSGEA. Este é detalhado na tabela 4.4. Os tipos GEREA e EA possuem vários atributos compostos que implementam listas. Nas tabelas 4.5 e 4.6 definem-se a entrada de cada elemento e as operações possíveis para cada lista de GEREA e EA, respectivamente. Como estas são operações padrão sobre listas, não estão completamente especificadas. Define-se apenas o domínio e a imagem de cada operação.

TABELA 4.2 - Tipos escalares usados na descrição do nível básico de SIMOO

Tipo	Semântica
bool	Pode assumir os valores T(true) ou F(false)
byte	Pode assumir valores pertencentes ao intervalo [0,255]
int	Pode assumir qualquer valor decimal inteiro
real	Pode assumir qualquer valor numérico, inteiro ou fracionário
string	Pode assumir qualquer seqüência de caracteres ASCII

TABELA 4.3 - Tipos compostos utilizados na camada de controle de elementos autônomos

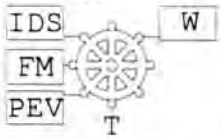
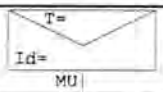
Tipo	Atributos	Símbolo
GEREA	IDS: lista, armazena identificadores de Eas presentes no sistema FM: lista, armazena os eventos programados para o futuro PEV: lista, armazena a relação dos EAs que ainda têm eventos pendentes para o tempo corrente em suas filas de mensagens. W: lista de EAs que se encontram suspensos aguardando msgs T: real, armzena o tempo de simulação corrente	
MSGCTRL	T: real, tempo para o qual a mensagem foi programada Id: string, identificador da mensagem de controle MU: MSGEA, mensagem de elemento autônomo	
EA	FM: lista, armazena as mensagens pendentes para o tempo corrente Pai: string, identificador do EA pai na hierarquia de agregações Id: string, identificador do EA WM: MSGEA, armazena mensagem aguardada ou recebida ou a constante NONE conforme o valor do atributo W. W: enumeração, indica se o EA está aguardando uma msg ou não. Pode assumir como valor as constantes: F (False) não está aguardando mensagem, WFN (Waiting For Now) aguardando mensagem para o tempo corrente, WFAT (Waiting For Any Time) aguardando mensagem para qualquer tempo ou RECV (Received) mensagem recebida e armazenada no atributo WM. TM: bool, indica se o EA encontra-se tratando uma mensagem ou não	

TABELA 4.4 - Detalhamento do tipo MSGEA

TIPO MSGEA		
Atributo	Tipo	Semântica
idorig	string	Identificador do EA que envia a mensagem
iddest	string	Identificador do EA que deve receber a mensagem
idmsg	string	Identificador da mensagem
pr	byte	prioridade da mensagem
p	lista	parâmetros da mensagem

TABELA 4.5 - Operações sobre as listas do gerenciador de elementos autônomos

Operações definidas sobre as tabelas do Gerenciador de Elementos Autônomos			
Tabela	**Elemento**	**Domínio**	**Operação**
IDS Armazena lista dos identificadores de EAs presentes no modelo	EIDS(Id,idpai) Id: string Idpai: string	IsIn: IDS x EIDS → bool Ins: IDS x EIDS → IDS Del: IDS x EIDS → IDS	IsIn(IDS,EIDS(d,idpai)) Ins(IDS,EIDS(d,idpai)) Del(IDS,EIDS(d,idpai))
FM Lista de mensagens programadas para tempos futuros	EFM(t,M) t: real M: MSGEA	Empty: FM → bool Ins: FM x EFM → FM Del: FM x EFM → FM Topo: FM → M TopTime: FM → real	Empty(FM) Ins(FM,EFM(t,M(A,B,Idm,pr,p))) Del(FM,EFM(t,M(A,B,Idm,pr,p))) Topo(FM) TopTime(FM)
PEV Armazena lista de EAs que possuem msgs pendentes	EPEV(Id,Idm) Id: string Idm: string	IsIn: PEV x EPEV → bool Ins: PEV x EPEV → PEV Del: PEV x EPEV → PEV	IsIn(PEV,EPEV(Id,Idm)) Ins(PEV,EPEV(Id,Idm)) Del(PEV,EPEV(Id,Idm))
W Armazena lista de EAs que se encontram suspensos aguardando mensagens	EW(Id) Id: string	IsIn: W x EW → bool Ins: W x EW → W Del: W x EW → W Empty: W → bool	IsIn(W,EW(id)) Ins(W,EW(id)) Del(W,EW(id)) Empty(W)

TABELA 4.6 - Operações sobre as listas de elemento autônomo

Operações Definidas sobre as tabelas de um Elemento Autônomo			
Tabela	**Entrada (Ent)**	**Domínio**	**Operação**
FM Armazena mensagens pendentes	EFM(M) M: MSGEA	Empty: FM → bool Ins: FM x EFM → FM Del: FM x EFM → FM Topo:FM → M	Empty(FM) Ins(FM,EFM(M(A,B,Idm,pr,p))) Del(FM,EFM(M(A,B,Idm,pr,p))) Topo(FM)

As interações entre os elementos autônomos são intermediadas pelo gerenciador de elementos autônomos. Efetivamente, elementos autônomos enviam mensagens de controle ao gerenciador requisitando as mais diversas tarefas tais como: programar o envio de uma mensagem para um elemento autônomo, criar ou remover instâncias de elemento autônomo, etc. As relações entre os tipos de dados presentes no nível de descrição básico podem ser vistas na figura 4.4.

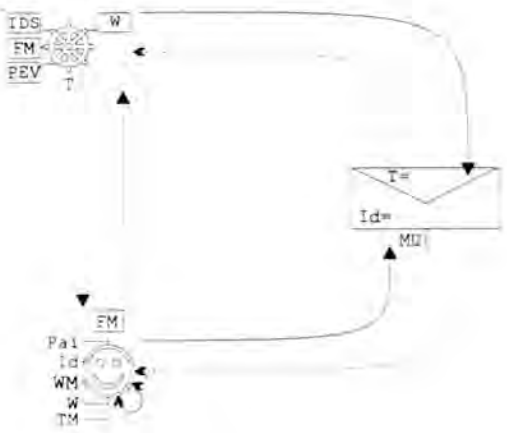


FIGURA 4.4 - Grafo de tipos do nível de descrição básico

As setas contínuas ligando o elemento autônomo ao gerenciador e consigo mesmo representam conhecimento. A seta tracejada indica um relacionamento entre tipos que especifica a existência de um protocolo de comunicação estabelecido entre os mesmos (observe que o tipo EA está conectado consigo mesmo indicando a possibilidade de instâncias de EA comunicarem-se entre si). As setas contínuas conectando o elemento autônomo e o gerenciador à mensagem de controle indicam que estes podem gerar este tipo de mensagem. Setas tracejadas conectando estes mesmos pontos indicam que eles estão aptos a receber mensagens de controle.

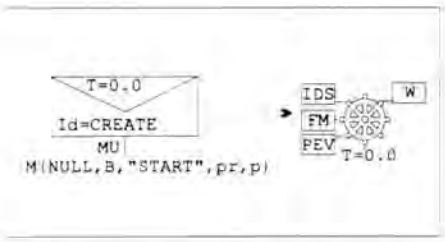


FIGURA 4.5 - Situação inicial de um modelo SIMOO

Para que uma simulação possa se iniciar é necessário que se estabeleçam algumas condições. É preciso que exista uma instância do gerenciador de elementos autônomos e que tal instância receba uma mensagem de controle do tipo "create" (veja seção 4.2.2.5, regra R11) solicitando a criação de um elemento autônomo. Este será o pai da hierarquia de agregações que compõe o modelo. A figura 4.5 apresenta essa situação.

4.2.2 O Conjunto de Regras

Uma vez definidos os tipos envolvidos e o estado inicial da simulação o comportamento dos elementos autônomos é determinado por um conjunto de regras. Estas regras podem ser organizadas em dois grupos. O primeiro descreve as interações entre os elementos autônomos e seu gerenciador. O segundo descreve a forma como cada elemento autônomo reage à chegada de cada uma das mensagens que particularizam o comportamento da entidade no modelo. SIMOO define o primeiro conjunto de regras. A especificação do segundo conjunto corresponde à construção do modelo de simulação.

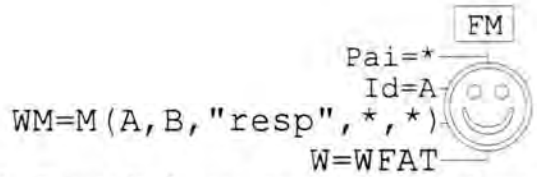


FIGURA 4.6 - Exemplo de representação dos valores dos atributos

As interações entre os elementos autônomos e seu gerenciador são governadas por um conjunto de 16 regras detalhadas a seguir. Na descrição destas regras, os tipos são instanciados indicando-se valores ou variáveis para seus atributos. Esta indicação é feita utilizando-se o símbolo de atribuição. Atributos cujo valor pode ser desconsiderado na aplicação de uma determinada regra são omitidos ou associados a um asterisco. Na figura 4.6 observa-se uma instância do tipo EA. O valor do campo "Pai" está sendo desconsiderado assim como os campos "prioridade" e "parâmetros" da mensagem (MSGEA) atribuída ao campo WM. O campo TM, cujo valor também pode ser desconsiderado, foi omitido.

4.2.2.1 Programação de mensagens

A primeira regra descreve a operação mais usual: a programação de uma mensagem para um tempo simulado maior ou igual ao corrente, tendo por destinatário um elemento autônomo qualquer. A programação de mensagens futuras é a forma disponibilizada por SIMOO para representar a programação de eventos. SIMOO não distingüe mensagens de eventos que são "simulados" por mensagens temporizadas.

Como pode-se observar pela figura 4.7, a regra R1 determina que a programação de uma mensagem implica na presença do emissor (Id = A) e do destinatário (Id = B). O emissor deve "conhecer" o destinatário (conhecer seu identificador) e seus tipos devem estar conectados (deve haver um protocolo de comunicação estabelecido entre os dois tipos de elemento autônomo). Finalmente, a mensagem deve ser programada para um tempo igual ou superior ao corrente[2]. A solicitação de envio da mensagem é feita ao gerenciador que armazena-a em sua fila de mensagens para enviá-la quando for oportuno.

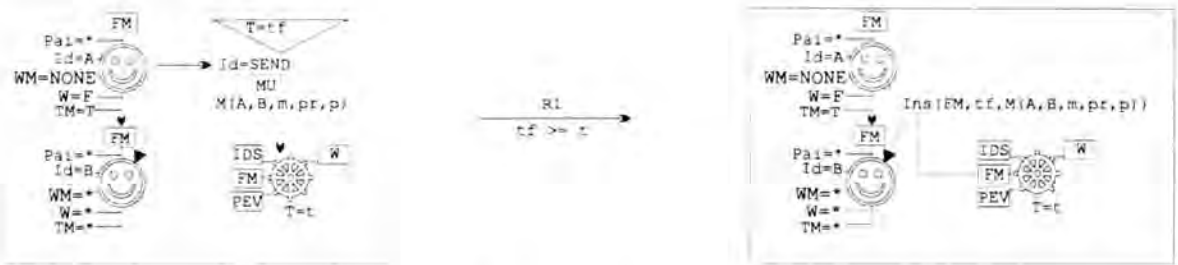


FIGURA 4.7 - Programando uma mensagem

4.2.2.2 Mecanismos de disparo de mensagens e avanço do relógio

As regras R2, R3 e R4 (figura 4.8) regulam o avanço do relógio de simulação e o envio das mensagens programadas.

[2]Isto é indicado pela expressão boleana tf > t, que deve ser verdadeira para que esta regra possa ser aplicada.

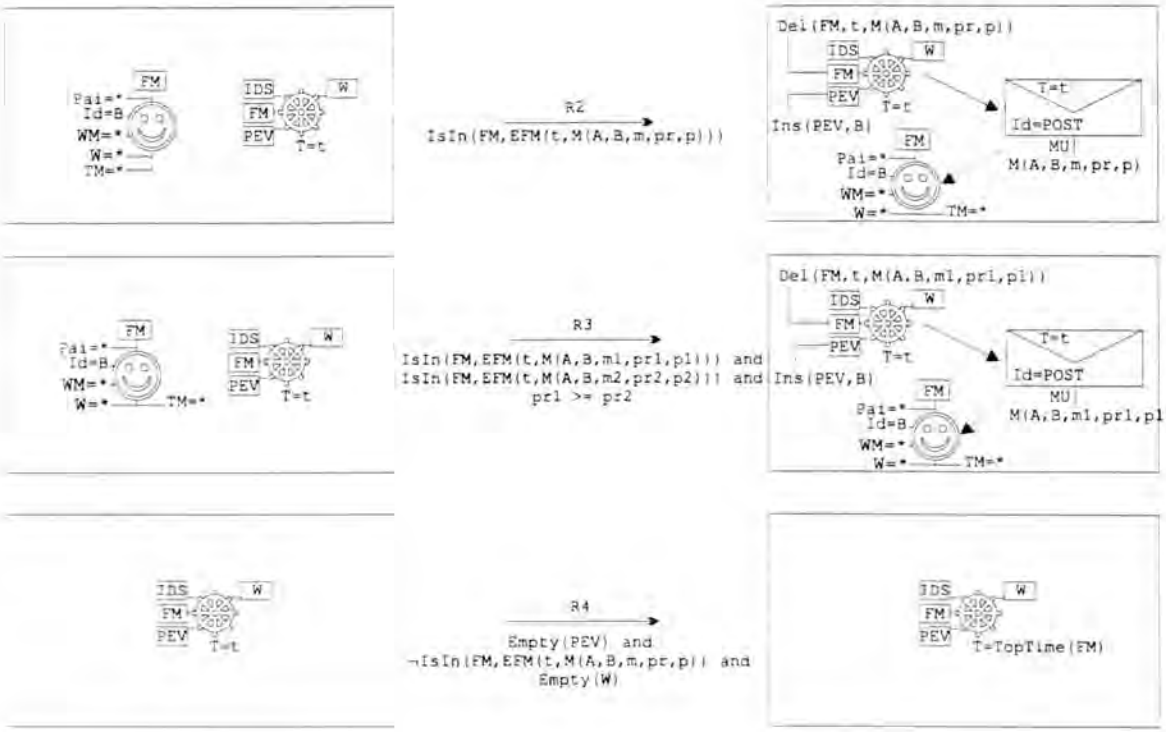


FIGURA 4.8 - Programando uma mensagem

Regra R2: quando houverem entradas na tabela de mensagens do gerenciador cujo atributo "T" (tempo) seja igual ao tempo corrente (atributo "T" de GEREA), as mesmas são removidas, uma por vez, da fila de mensagens do gerenciador e enviadas aos destinatários correspondentes.

Regra R3: se, na fila de mensagens do gerenciador, houverem duas mensagens programadas para o mesmo tempo e destinatário, será enviada primeiro a que tiver maior prioridade. Se as prioridades forem iguais segue-se qualquer ordem.

Regra R4: por fim, o relógio de simulação avança quando não houverem mais mensagens programadas para o tempo corrente, todos os elementos autônomos tiverem acabado de processar as mensagens que se encontravam pendentes em suas próprias filas e nenhum elemento autônomo estiver aguardando por uma mensagem "FORNOW".

Elementos autônomos podem enviar mensagens síncronas. Neste caso, após o envio da mensagem o elemento autônomo entra em estado de espera pela resposta correspondente. Mensagens síncronas podem ser de dois tipos: "FORNOW" quando a resposta deve ser instantânea em termos de tempo de simulação e "FORANYTIME" quando não existir tal obrigatoriedade. O tipo de espera é indicado pelo valor do atributo "W" (WFN - Wait For Now e WFAT - Wait For Any Time). Sendo assim, o relógio simulado não pode avançar sem que todas as mensagens "FORNOW" tenham sido respondidas. Se eventualmente, por falha no modelo, isto ocorrer, a execução da simulação é suspensa e o sistema reporta uma mensagem de erro.

A sistemática de avanço do relógio e disparo das mensagens permite deduzir o grau de paralelismo permitido por SIMOO. O gerenciador dispara todas as mensagens

previstas para o tempo corrente. Como os elementos autônomos são máquinas seqüenciais, consumirão as mensagens depositadas em suas filas de forma seqüencial, porém de forma instantânea do ponto de vista do tempo simulado. Se as mensagens tratadas programarem novas mensagens para o tempo corrente estas são imediatamente disparadas. Sendo assim, o paralelismo é obtido na execução paralela das mensagens programadas para o mesmo tempo para os diferentes elementos autônomos, uma vez que todos estão sincronizados pelo tempo. A opção por uma estratégia conservativa foi feita em função da complexidade global do sistema. A estratégia pode ser alterada, entretanto, modificando-se algumas poucas regras.

4.2.2.3 Enfileirando mensagens em um elemento autônomo

A forma pela qual as mensagens são recebidas pelos EAs é descrita pelas regras de R5 a R9 (figura 4.9).

Durante o tratamento de uma mensagem, um elemento autônomo pode entrar em estado de espera (veja seção 4.2.2.2). O estado em que o elemento autônomo se encontra influencia a maneira como as mensagens são recebidas. Se o mesmo não se encontra em estado de espera e recebe uma nova mensagem, esta é imediatamente enfileirada para que possa ser tratada assim que possível (regra R5). Se estiver em estado de espera é necessário analisar diferentes possibilidades.

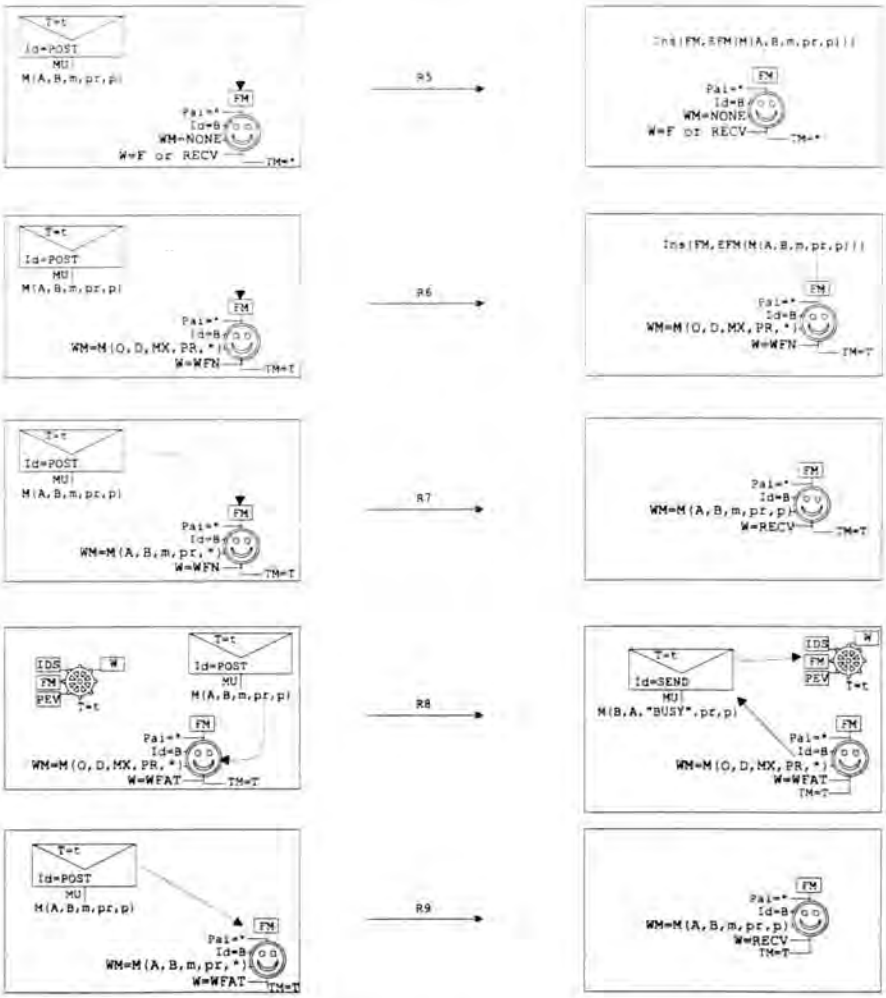


FIGURA 4.9 - Recebimento de uma mensagem por um EA

Se o elemento autônomo se encontra no estado de espera "FORNOW" e a mensagem que chega não é "liberadora", isto é, não é a mensagem aguardada, a mesma pode ser enfileirada porque pode-se garantir que será tratada ainda durante o tempo corrente (regra R6). Se, por outro lado, a mensagem que chega for a mensagem liberadora, então esta não é enfileirada. A mensagem é anotada (atributo "WM") e o estado do elemento autônomo passa a ser "RECV" (atributo "W"), ou seja, mensagem resposta recebida (regra R7). O tratamento da mensagem responsável pelo estado de espera pode prosseguir. O estado do elemento autônomo deixará de ser "RECV" ao final do tratamento da mensagem corrente ou se for feita nova solicitação de aguardo por mensagem antes do término de seu tratamento.

Se o elemento autônomo estiver esperando por uma mensagem "FORANYTIME" (atributo "W"=WFAT), então o enfileiramento de novas mensagens não é permitido, visto que não existe garantia de que serão atendidas durante o tempo corrente (regra R8). Neste caso a mensagem recebida se perde, porém uma notificação é enviada para o elemento autônomo emissor.

Finalmente, se o EA se encontra em estado de espera "FORANYTIME" e a mensagem que chega é a esperada (regra R9), então o tratamento é semelhante ao da regra 7.

4.2.2.4 Cancelamento de eventos

O cancelamento de eventos é descrito pela regra R10 (figura 4.10). Qualquer elemento autônomo pode cancelar uma mensagem por ele programada para um tempo futuro. Não é possível cancelar mensagens programadas para o tempo corrente[3]. Para tanto basta enviar a solicitação adequada ao gerenciador desde que a mensagem ainda não tenha sido disparada. Mensagens programadas para o tempo corrente não podem ser canceladas porque não se tem certeza da ordem em que os eventos são processados em um instante do tempo simulado e o evento a cancelar pode já ter sido tratado.

FIGURA 4.10 - Cancelamento de evento

4.2.2.5 Criação e remoção de elementos autônomos

SIMOO impõe uma modelagem hierárquica. No nível mais alto de abstração existe uma entidade única, o sistema sendo simulado, a qual é necessariamente refinada nos níveis inferiores constituindo uma hierarquia agregações. Esta estratégia facilita o reuso das entidades bem como permite que se aprofunde o grau de detalhamento na medida do necessário.

---

[3]Isto é indicado pela expressão boleana tf > t, que deve ser verdadeira para que esta regra possa ser aplicada.

A criação de um elemento autônomo é descrita pela regra R11 (figura 4.11). Observe que a criação é feita através do envio de uma mensagem do tipo "CREATE" para o gerenciador de elementos autônomos. Note, ainda, que o elemento autônomo criado irá considerar como elemento autônomo "pai" (agregador) aquele que solicitou a criação (veja o valor do campo "pai" do elemento autônomo criado no lado direito da regra). A criação de um elemento autônomo implica ainda no seu cadastramento junto ao gerenciador (tabela IDS) e a inserção de uma mensagem "START" na fila de mensagens do elemento autônomo recém criado. O valor dos parâmetros desta mensagem são determinados pelo elemento que solicitou a criação através dos parâmetros da mensagem "CREATE".

As regras R12 e R13 (figura 4.11) descrevem a remoção de elementos autônomos. Um elemento autônomo solicita a destruição de outro pelo envio da mensagem "SIMOO_QUIT" ao gerenciador. A regra 12 indica que quando um elemento autônomo recebe uma mensagem "SIMOO-QUIT", ele dispara mensagens semelhantes para todos os seus agregados e assim recursivamente (o ciclo de vida das entidades agregadas é dependente do ciclo de vida de seu agregador, logo, quando da remoção de um elemento autônomo, seus agregados devem ser removidos automaticamente). A regra 13 diz que um elemento autônomo que recebeu uma mensagem do tipo "SIMOO_QUIT" pode ser removido quando não tiver agregados ou seus agregados tiverem sido removidos.

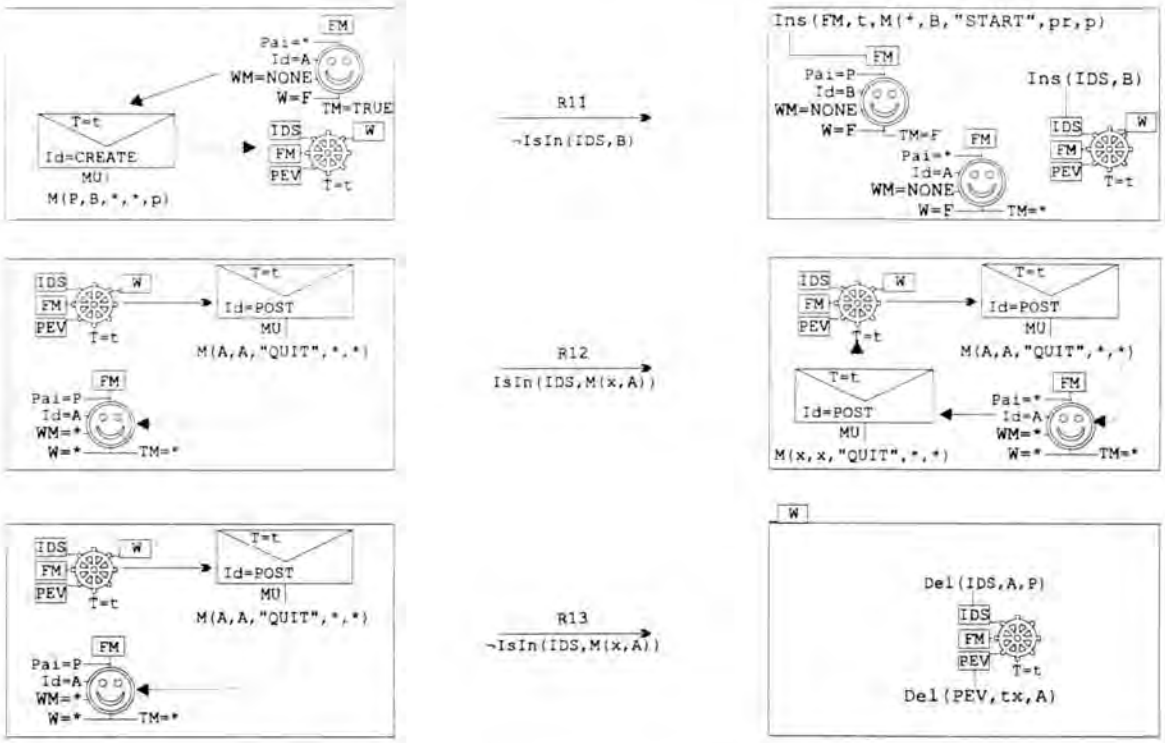


FIGURA 4.11 - Criação e destruição de EAs

4.2.2.6 Entrando em estado de espera

Por fim, as regras R14 a R16 (figura 4.12) definem o comportamento dos elementos autônomos no que se refere a interrupções no tratamento de uma mensagem.

Um elemento autônomo pode executar no máximo duas atividades em paralelo: tratar uma mensagem (já se viu que o tratamento das mensagens é seqüencial) e receber novas mensagens (enfileirá-las) para tratamento futuro. Por vezes, durante o tratamento de uma mensagem, um elemento autônomo necessita que a coleta de uma informação ou a solicitação de um serviço junto a outro elemento autônomo seja feita de forma síncrona. Isso significa que o tratamento da mensagem corrente deve ser suspenso até que a informação tenha sido recebida ou o serviço executado (veja seção 4.2.2.3). Outras vezes a interrupção pode ser por um período de tempo conhecido de maneira a simular a passagem do tempo no modelo (tempo de atendimento, por exemplo). Enquanto o tratamento da mensagem estiver suspenso, o elemento autônomo pode apenas tratar o recebimento de mensagens.

Como já foi comentado anteriormente, existem dois tipos de interrupções: interrupção "instantânea" (FORNOW) e interrupção "demorada" (FORANYTIME).

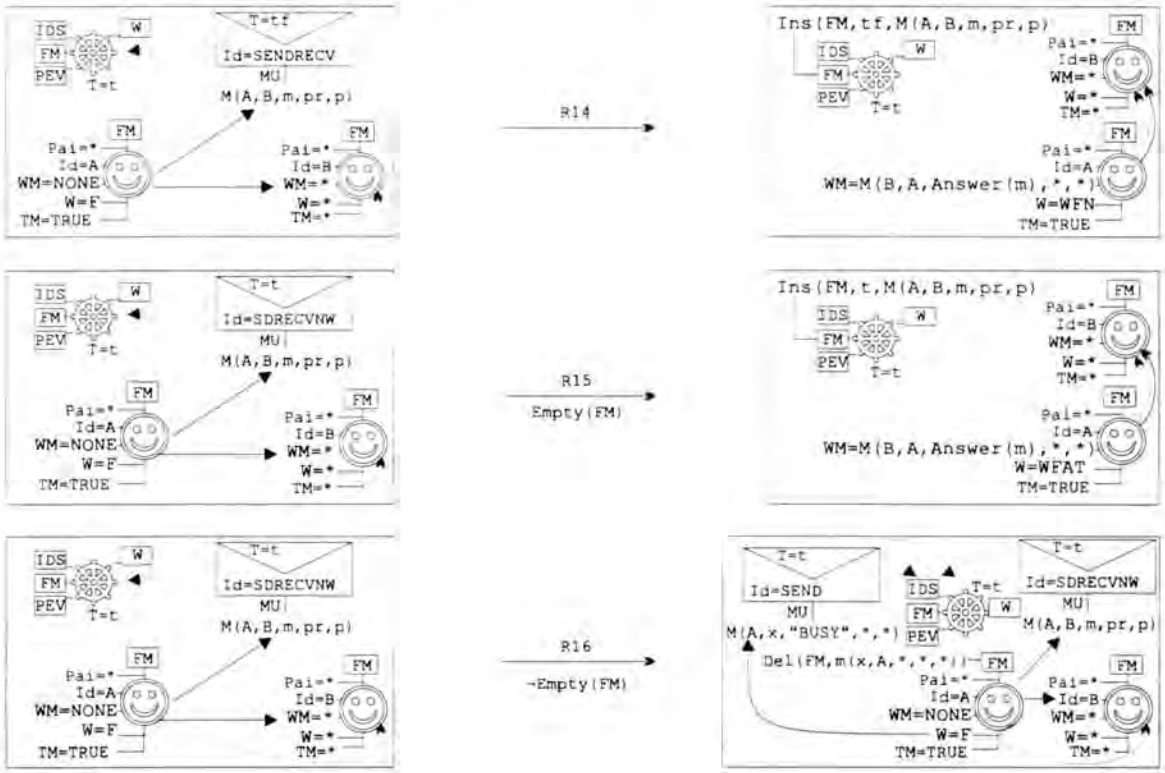


FIGURA 4.12 - Solicitações de pausa

A regra R14 demonstra a programação de uma mensagem "FORNOW". Gera-se uma mensagem normal para o elemento destino enquanto que o elemento emissor passa para o estado "WFN". O elemento emissor permanecerá neste estado até que chegue a mensagem aguardada conforme estabelecido na regra R7.

As regras R15 e R16 demonstram a programação de uma solicitação "FORANYTIME". Se a fila de mensagens do elemento autônomo solicitante estiver vazia, então o procedimento é semelhante ao descrito na regra R14 com a diferença que o elemento autônomo passa para o estado "WFAT" (regra R15). Se, por outro lado, houverem mensagens pendentes na fila do elemento autônomo, então é necessário que

elas sejam canceladas pois não existe mais garantia de que as mesmas serão tratadas durante o tempo corrente. Para tanto todos os elementos emissores dessas mensagens devem ser notificados (regra 16).

Solicitações de aguardo por um tempo determinado são efetivadas programando-se uma mensagem para o próprio elemento autônomo solicitante para o tempo futuro correspondente ao tempo atual mais o tempo de suspensão pretendido. O elemento autônomo entra, então, em estado de espera pela mensagem que ele mesmo programou.

4.2.3 Descrevendo um modelo

A seção 4.2.2 enumera o conjunto de regras que descrevem as interações entre os elementos autônomos e o gerenciador. Para se construir um modelo de simulação, usando-se a gramática de grafos apresentada, é necessário estender o conjunto inicial de regras descrevendo as reações de cada tipo de elemento autônomo às mensagens previstas. Embora a versão atual de SIMOO não permita este tipo de especificação, julgou-se interessante mostrar como seria esse procedimento.

A figura 4.13 apresenta o exemplo de um tipo de elemento autônomo que possui um atributo de tipo inteiro chamado "Cont". Sempre que for retirada da fila de mensagens deste tipo de elemento autônomo a mensagem "INC", o valor do atributo "Cont" será incrementado de 1.

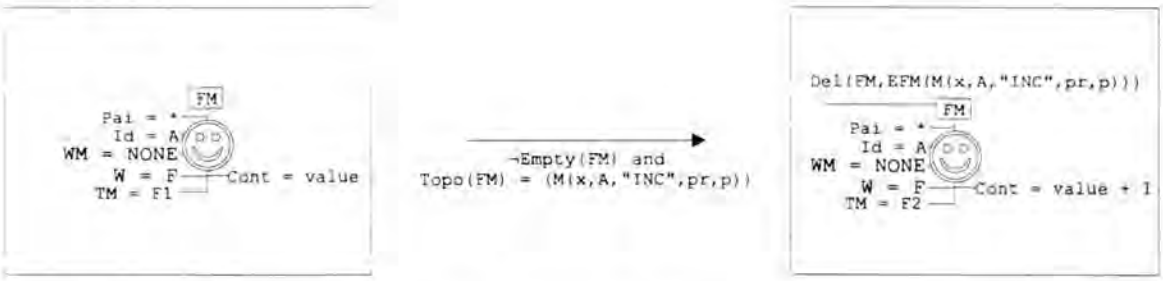


FIGURA 4.13 - Exemplo de descrição de comportamento específico

Observe, ainda, que durante o tratamento da regra o valor do atributo TM passa a ser TRUE e volta a ser FALSE no final. A regra não explicita o instante em que o atributo TM recebe o valor TRUE porque é uma mensagem de tratamento muito simples, instantâneo, que pode ser facilmente descrito através de uma única regra. Esta situação é um exemplo do que foi explicado na seção 4.2, isto é, apesar do valor de TM ser o mesmo antes e depois da aplicação da regra, ele se altera durante sua aplicação. Isso é representado pelos índices ao lado do valor de TM (F_1 e F_2) nos dois lados da regra. Os índices indicam que apesar do valor ser o mesmo FALSE ele se alterou durante a aplicação da regra.

A figura 4.14 procura incrementar a complexidade do exemplo. Ao receber a mensagem "INC", o elemento autônomo deverá somar 1 ao valor do atributo Cont, na seqüência somar mais 2 ao valor do atributo Cont e finalmente somar mais 1 ao valor do atributo Cont novamente. Embora este exemplo não faça muito sentido (seria mais simples incrementar Cont de 5 diretamente) optou-se por usar operações simples e semelhantes para destacar a forma de se representar a seqüencialidade das operações.

Observe o uso de uma notação específica para representar a ordem em que os passos devem ser executados.

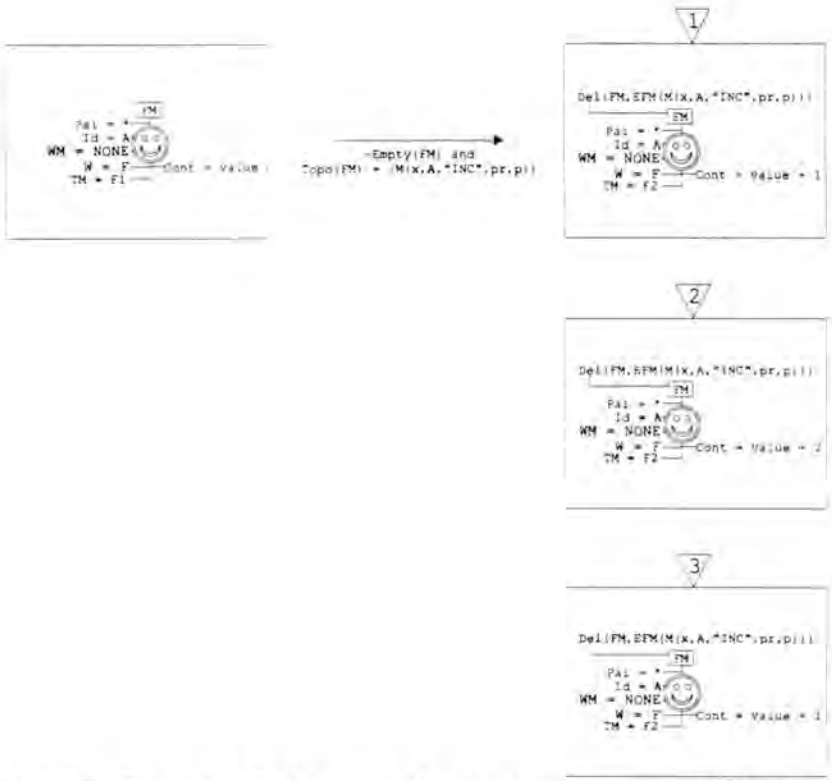


FIGURA 4.14 - Exemplo de descrição de comportamento específico que exige tratamento seqüencial

4.2.4 Monitores

Monitores são uma categoria especial de elemento autônomo que tem a capacidade de observar a troca de mensagens entre os demais elementos autônomos do modelo. Esse tipo de elemento autônomo visa a construção de elementos responsáveis por tarefas ligadas aos experimentos de simulação e não ao modelo em si tais como coleta de estatísticas e controle de visualização e interação com o usuário. Desta forma garante-se uma maior independência de domínios mantendo-se pura a descrição do modelo propriamente dito e facilitando o reuso tanto das entidades como destes elementos de controle.

Um elemento monitor é um elemento autônomo comum. Para se tornar um monitor, o mesmo deve se cadastrar junto ao gerenciador usando a mensagem SIMOO_CATALOG_MONITOR. A partir desse momento passa a receber uma cópia de todas as mensagens trocadas entre os elementos sendo monitorados. Para poder distingüir as mensagens monitoradas das mensagens destinadas para o próprio monitor, as mensagens monitoradas são antecedidas do símbolo "%". Um elemento autônomo qualquer cadastra-se junto ao gerenciador para ter suas mensagens monitoradas por um determinado monitor através da mensagem "SIMOO_START_MONITOR". A interrupção no processo de monitoração pode ser solicitada pela mensagem "SIMOO_STOP_MONITOR".

4.2.5 Mensagens padrão

A seção 4.2.2 descreveu as principais regras que controlam o ciclo de vida de um elemento autônomo e do gerenciador de elementos autônomos. Além destas regras, a definição destes elementos prevê um conjunto de "mensagens padrão" para as quais o comportamento é pré-determinado. São mensagens de elemento autônomo (MSGEA) que são exploradas pela interface padrão (veja capítulo 3) e podem ser exploradas pelo usuário projetista na construção do modelo.

Por uma questão de convenção, todas as mensagens padrão tem seu identificador iniciado pelo substring "SIMOO_". Uma destas mensagens, "SIMOO_QUIT", por sua importância, foi descrita nas regras R12 e R13 (seção 4.2.2.6). As demais, por questão de simplicidade são listadas e descritas textualmente na tabela 4.7.

TABELA 4.7 - Mensagens padrão tratadas no nível básico

Origem	Destino	Identificador	Semântica/Funcionamento
EA	EA	SIMOO_ISBUSY?	Questiona o elemento autônomo destino se o mesmo se encontra ocupado (TM = WFAT)
EA	GEREA	SIMOO_ISALIVE?	Verifica se determinada instância de elemento autônomo não foi removida do modelo.
EA	EA	SIMOO_INTERRUPT	Força um elemento autônomo a sair do estado WFAT e indica quem solicitou essa interrupção
EA	GEREA	SIMOO_MONITOR	Indica ao gerenciador que o elemento emissor passa a ser um monitor
EA	GEREA	SIMOO_START_MONITOR	Indica ao gerenciador que o elemento emissor passa a ser monitorado
EA	GEREA	SIMOO_STOP_MONITOR	Indica ao gerenciador que o elemento emissor tem a monitoração suspensa

4.3 O nível de interação com elementos de interface

O nível de elementos de interface disponibiliza os elementos que são utilizados para a interação com o usuário, seja para a entrada de valores de parâmetros seja para a visualização de resultados, seja para a modificação da estrutura do comportamento do modelo.

4.3.1 Características gerais

Elementos de interface assemelham-se a elementos autônomos na medida em que são autônomos e comunicam-se por mensagens. As principais diferenças residem no fato de que possuem representação visual e não estão sincronizados em relação ao relógio simulado.

A representação visual de um elemento de interface permite que o usuário de um modelo interaja com o mesmo. Essa interação pode-se restringir à simples observação de resultados até a alteração de parâmetros, comportamento de entidades ou do relógio de simulação, dependendo da forma e do propósito do elemento de interface.

Por permitirem a interação com o usuário, elementos de interface não podem ser sincronizados em relação ao relógio simulado e sim em relação ao relógio real. Por utilizar o mesmo modelo de troca de mensagens, não existe nenhum impedimento em relação ao usuário interagir com o modelo através dos elementos de interface, com a simulação em andamento. A dificuldade será determinar em que momento do tempo

simulado os resultados dessa interação serão passados ao modelo. Por esta razão é mais interessante interagir com o modelo estando a simulação momentaneamente suspensa.

4.3.2 Tipos e relacionamentos

O nível de elementos de interface acrescenta três novos tipos aos descritos no nível básico: ELINTF, que descreve um elemento de interface, GERELINTF, que descreve o gerenciador de elementos de interface, e MSGELINTF, que descreve uma mensagem de controle entre elementos de interface e seu gerenciador e não as mensagens que circulam entre os elementos de interface, de forma semelhante ao tipo MSGCTRL (veja tabela 4.8).

TABELA 4.8 - Tipos adicionados no nível de elementos de interface

Tipo	Símbolo
ELINTF Elemento de Interface	
GERELINTF Gerenciador de elementos de interface	
MSGELINTF Mensagem de elemento de interface	

A figura 4.15 apresenta o grafo de tipos que inclui os tipos inseridos neste nível. Observe que assim como no caso dos elementos autônomos, a troca de mensagens entre elementos de interface é feita através de seu gerenciador específico (GERELINTF). Embora elementos de interface possam conhecer (armazenar os identificadores) de elementos autônomos e vice versa, a troca de mensagens entre ambos é intermediada pelos dois gerenciadores. Isso é necessário para o devido ajuste entre as mensagens não temporizadas que envolvem os elementos de interface e as mensagens temporizadas que circulam entre os elementos autônomos.

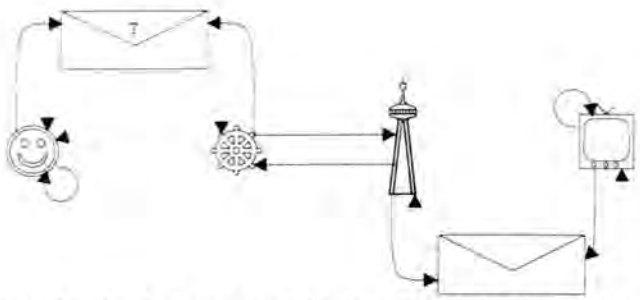


FIGURA 4.15 - Grafo de tipos incluindo o nível de elementos de interface

4.3.3 Regras

Excetuando-se o fato de possuírem representação visual, as funcionalidades de um elemento de interface são semelhantes às dos elementos autônomos. A riqueza dos aspectos visuais e de interação com o usuário depende do ambiente operacional onde eles forem implementados.

A troca de mensagens entre elementos de interface é feita com o auxílio de mensagens de controle trocadas com o gerenciador de elementos de interface de forma semelhante a que ocorre com os elementos autônomos. Este tipo de comunicação se faz necessário nos casos em que um elemento de interface cria outro, por exemplo, para a solicitação de um conjunto de valores ou para a exibição de informações adicionais. A comunicação com elementos autônomos é intermediada pelos dois gerenciadores visto que existe necessidade de se compatibilizar os diferentes tipos de mensagens.

Quando um elemento autônomo deseja enviar uma mensagem para um elemento de visualização, procede da mesma forma como se fosse enviar a mensagem para outro elemento autônomo. O gerenciador de elementos autônomos se encarrega de retê-la até o momento do tempo simulado para o qual a mensagem foi prevista e, então, a repassa para o gerenciador de elementos de interface que a entrega para seu destinatário.

Quando a situação é inversa, ou seja, um elemento de interface deseja enviar uma mensagem para um elemento autônomo, a solicitação é feita para o gerenciador de elementos de interface que a transmite para o gerenciador de elementos autônomos. Esse acrescenta o tempo corrente como "time-stamp" e a insere em sua fila de mensagens para tratamento normal.

Por fim, elementos de interface são capazes de solicitar informações ao seu gerenciador, referentes ao conjunto de elementos autônomos e ao próprio conjunto de elementos de interface. Estas solicitações referem-se às informações mantidas pelos gerenciadores tais como categorias de elementos autônomos presentes no modelo, instâncias de cada categoria, tempo corrente, mensagens programadas para o futuro, etc. Podem ser solicitadas também informações sobre a estrutura de cada categoria de elemento autônomo (tipo e conteúdo dos atributos, mensagens a que responde, etc). Estas informações são fornecidas pelo próprio gerenciador de elementos de interface, ou pelo gerenciador de elementos autônomos a partir de solicitação do primeiro ou pelos próprios elementos autônomos a partir de solicitação do gerenciador. Este tipo de solicitação é importante, para que o sistema possa permitir ao usuário investigar ou alterar qualquer aspecto do modelo durante a execução.

Uma vez que os elementos de interface não são sincronizados pelo relógio simulado, suas mensagens são sempre repassadas instantaneamente. De forma semelhante aos elementos autônomos, elementos de interface possuem uma fila de mensagens e as processam seqüencialmente na medida de suas possibilidades.

Por estarem permanentemente disponíveis para a interação com o usuário, elementos de interface não podem fazer consultas síncronas. Para contornar esse problema existem os filtros de mensagem. No caso de consulta síncrona, um elemento de interface envia uma mensagem ao destinatário (o protocolo deve garantir que este providenciará uma resposta) e uma mensagem ao gerenciador solicitando filtragem de mensagens. O mecanismo de filtragem só faz com que o gerenciador de elementos de interface só permita que uma determinada mensagem chegue ao elemento que tem o filtro associado. Qualquer outra mensagem é recusada e uma mensagem de erro enviada ao emissor. Desta forma, um elemento de interface pode simular uma comunicação síncrona sem ter seu processamento efetivamente suspenso. O filtro é desativado

automaticamente na chegada da mensagem aguardada. A falta de sincronismo torna o abastecimento da fila de mensagens de um elemento de interface uma tarefa bastante simples. Toda a mensagem que chega pode ser enfileirada para tratamento desde que não haja restrição relativa à filtragem.

Quando não houverem mais mensagens a serem tratadas, elementos de interface entram em estado passivo à espera de uma ação do usuário ou chegada de uma mensagem nova. Por uma questão de simplicidade, todas as ações do usuário são convertidas em mensagens como se tivessem sido enviadas por um elemento autônomo, padronizando o tratamento.

A criação ou destruição de elementos de interface é feita por seu gerenciador a partir de solicitações originárias de elementos autônomos ou dos próprios elementos de interface. Solicitações provenientes de elementos autônomos dizem respeito aos aspectos de visualização previstos no próprio modelo, seja integrado na descrição do comportamento das entidades, seja em um elemento de monitoração. Solicitações provenientes dos próprios elementos de interface são relativas a construção da própria interface de controle ou por comando do usuário através da interface de controle para a associação de uma representação visual a um atributo de uma entidade, conforme descrito na seção 1.7.

Quando ocorrer uma solicitação de destruição, um elemento de interface apenas encerra o processamento da mensagem corrente. Mensagens pendentes em sua fila não são tratadas.

Em geral, elementos de interface são criados por elementos autônomos. Analisando-se a figura 4.5, que descreve a situação inicial de um modelo SIMOO, percebe-se que não existem elementos de interface presentes, apenas um elemento autônomo. Desta forma, pelo menos o primeiro dos elementos de interface a ser criado durante a execução do modelo será criado por um elemento autônomo. Os elementos de interface que compõem a interface de controle são criados pelo elemento autônomo de mais alto nivel na hierarquia do modelo. Código para tanto é acrescentado pela ferramenta de modelagem durante a geração do modelo executável.

Esse fato é significativo porque o ciclo de vida dos elementos de visualização permanece sempre atrelado ao ciclo de vida do elemento autônomo que o criou (de forma direta ou indireta). O identificador do elemento autônomo criador fica registrado no identificador do elemento de interface. A regra de formação de identificadores de elementos de interface é semelhante a regra de formação de identificadores de elementos autônomos. O identificador do elemento criador antecede sua própria denominação (veja figura 4.16). Identificadores de elementos de interface iniciam-se pelo símbolo "$" de maneira a diferenciá-los de identificadores de elementos autônomos.

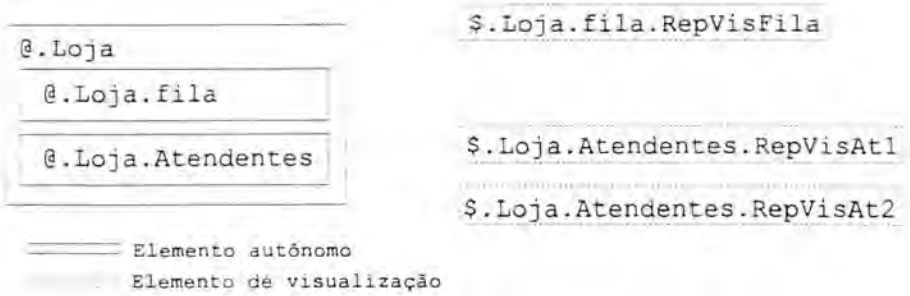


FIGURA 4.16 - Identificadores de elementos de interface

A maneira de compor os identificadores dos elementos de interface permite o reuso de um tipo de elemento autônomo que cria instâncias de elementos de visualização evitando problemas de duplicidade de identificadores no mesmo contexto.

4.4 O nível de paradigmas de simulação

4.4.1 Introdução

A principal característica do modelo de simulação de SIMOO é permitir ao usuário projetista do modelo selecionar o paradigma de simulação mais adequado à descrição de cada uma das entidades componentes do modelo, ou seja, à descrição de cada um dos tipos de elementos autônomos que farão parte do modelo. Para tanto disponibiliza um conjunto de paradigmas básicos a partir dos quais podem ser derivados outros. Estes são acrescidos de regras de compatibilização de maneira que possam ser usados em conjunto.

Os paradigmas suportados por SIMOO seguem o esquema apresentado na seção 3.7. Atualmente estão implementadas duas abordagens para a descrição dos eventos que definem alterações no estado do sistema (orientação a eventos e orientação a processos) e duas para descrever a comunicação entre as entidades (comunicação por mensagens e comunicação por portas). As possíveis combinações entre estas abordagens geram quatro possibilidades:

- entidades orientadas a eventos com comunicação por mensagens;
- entidades orientadas a eventos com comunicação por portas;
- entidades orientadas a processos com comunicação por mensagens;
- entidades orientadas a processos com comunicação por portas.

SIMOO deriva, a partir do elemento autônomo básico, quatro tipos de elementos autônomos, um para cada possibilidade. Cada um destes tipos torna visível para o usuário apenas aqueles recursos necessários para o paradigma pretendido. A partir destes é que o usuário projetista deriva os elementos autônomos que representam as entidades do modelo. As seções seguintes descrevem as características de cada uma das abordagens.

4.4.2 Elementos autônomos orientados a eventos x elementos autônomos orientados a processos

Elementos autônomos orientados a eventos têm seu comportamento descrito através de um conjunto de rotinas de eventos. Rotinas de eventos são consideradas

instantâneas em termos de tempo de simulação e por essa razão não podem conter instruções que impliquem em estender sua execução durante o avanço do relógio. Sendo assim, elementos autônomos orientados a eventos podem programar mensagens para o tempo corrente, podem programar mensagens para o futuro e podem fazer comunicações síncronas do tipo "FORNOW", porém não podem fazer comunicações síncronas do tipo "FORANYTIME". A descrição do comportamento de entidades descritas com esse tipo de elemento autônomo costuma ser quebrada em um conjunto de rotinas de eventos.

Elementos autônomos orientados a processos, por outro lado, têm seu comportamento descrito em uma única rotina de processo. Uma rotina de processo pode permanecer executando por várias unidades de tempo e por essa razão pode fazer comunicações síncronas do tipo "FORANYTIME". Se por um lado o uso de rotinas de processo permite agrupar a descrição do comportamento da entidade, é preciso que sejam tomados certos cuidados no atendimento de solicitações provenientes de outros elementos autônomos.

Como já foi visto, um elemento autônomo básico recusa o enfileiramento de mensagens quando se encontra em estado de espera devido a uma comunicação síncrona do tipo "FORANYTIME". As mensagens que chegam enquanto o elemento autônomo está tratando uma mensagem são enfileiradas, porém qualquer ocorrência de comunicação síncrona do tipo "FORANYTIME" provoca o cancelamento das mensagens pendentes.

Em uma rotina de processo aparecem muitas solicitações de comunicação síncrona do tipo "FORANYTIME", pois somente assim a rotina consegue permanecer ativa à medida que o relógio de simulação avança. Desta forma, dificilmente algum elemento autônomo consegue fazer solicitações a um elemento autônomo que está executando uma rotina de processo pois o elemento estará freqüentemente ocupado esperando o resultado de uma comunicação síncrona do tipo "FORANYTIME".

Em princípio estas limitações estão corretas pois se um elemento autônomo é seqüencial e está ocupado não pode atender solicitações externas. Ocorre que em muitas situações outras entidades precisam ter conhecimento do estado de uma entidade orientada a processos para tomar suas próprias decisões, ou simplesmente têm a necessidade de interromper sua execução por qualquer razão. Para estes casos existem duas alternativas. A primeira é interromper o estado de bloqueio do elemento autônomo que se encontra ocupado usando a mensagem "SIMOO_INTERRUPT" descrita na seção 4.2.3. A segunda é se valer do fato de que todo elemento autônomo provê automaticamente um conjunto de mensagens de questionamento sobre seus atributos (na versão atual essa característica é obtida usando-se um meta-objeto associado ao elemento autônomo). Estas mensagens não alteram o estado do elemento autônomo permitindo apenas que o valor de seus atributos seja consultado. Desta forma tais consultas são consideradas exceções e tratadas pelo elemento autônomo mesmo quando ele se encontra ocupado.

4.4.3 Elementos autônomos que se comunicam por mensagens x elementos autônomos que se comunicam por portas

A comunicação por mensagens é extremamente simples: ocorre aos pares e exige apenas que o elemento autônomo emissor da mensagem tenha conhecimento do identificador do elemento autônomo destinatário.

Quando se utilizam portas, por outro lado, o elemento autônomo só tem conhecimento dos identificadores de suas próprias portas. As portas podem ser de entrada (permitem a chegada de mensagens) ou de saída (permitem o envio de mensagens). Portas de saída podem ser conectadas a uma ou mais portas de entrada facilitando o envio de uma mensagem para um grupo de entidades.

Portas são conectadas através de mensagens. O elemento autônomo agregador de um conjunto de elementos autônomos é em geral o responsável por conectar as portas de seus agregados. Todo o componente possui uma porta implícita que permite a comunicação com seu pai. Para a conexão de portas existem as mensagens "SIMOO_CONNECT", "SIMOO_ISCONNECTED" e "SIMOO_DISCONNECT" para conectar, testar a conexão e desfazer uma conexão, respectivamente.

Para manter a compatibilidade entre os paradigmas, admite-se que um elemento autônomo que se comunique por mensagens envie uma mensagem para uma porta de entrada de um que se comunica por portas. Da mesma forma admite-se que uma porta de saída seja conectada diretamente a um elemento que se comunique por mensagens. Como elementos que se comunicam por portas não prevêem identificadores pelas mensagens já que apenas um tipo de mensagem circula por cada porta, no caso de uma porta de saída ser conectada a um elemento que se comunica por mensagens o nome da porta será usado como identificador da mensagem.

5 Representação Gráfica da Estrutura Estática do Modelo

5.1 Introdução

Entre os objetivos propostos para este trabalho está a possibilidade de se especificar modelos de simulação usando o paradigma de objetos através de uma notação que possa ser aproveitada em uma etapa posterior de implementação do sistema de controle das entidades reais. Some-se a isso o fato de que a notação utilizada deve ser completa o suficiente para que a partir da mesma possam ser gerados modelos executáveis.

Tendo em mente os requisitos propostos, faz-se necessário que características específicas de modelagem de sistemas de simulação discretos possam ser expressas. Entre estas características destaca-se o paradigma de simulação adotado para cada entidade, visto que sua escolha (portas ou mensagens, processos ou eventos, etc.) influencia diretamente a maneira como o comportamento das entidades é descrito e a forma como trocam informações.

Uma vez que os diagramas propostos em metodologias orientadas a objetos visam, em sua grande maioria, a especificação de requisitos e análise de sistemas [PER96] ou objetivam a documentação de programas orientados a objetos [WAS90] optou-se por estender as notações existentes. Em sua versão atual, a notação utilizada por SIMOO permite a representação gráfica da estrutura estática de um sistema, especificada através de um diagrama de classes e um diagrama de instâncias, enquanto a parte dinâmica é gerada a partir do preenchimento de formulários e da programação em C++ dos métodos de cada objeto.

Este capítulo descreve a notação utilizada para descrever a estrutura estática do modelo. A seção 5.2 apresenta o diagrama de classes e a seção 5.3 descreve o diagrama de instâncias.

5.2 O Diagrama de classes

Um modelo SIMOO é construído de maneira hierárquica, sendo que diversos tipos de hierarquias podem ser identificados: hierarquia de agregação, de herança, etc. Classes definidas em um determinado nível são consideradas componentes da classe de mais alto nível que está sendo detalhada. Um modelo SIMOO é, desta forma, representado por uma única classe que inclui todas as demais.

5.2.1 Classes e subdiagramas

Cada classe é representada graficamente por um retângulo, o qual é sub-dividido em duas seções (ver fig. 5.1):

- a seção do lado direito contém um conjunto de três ícones que indicam as abordagens que compõem o paradigma de simulação utilizado para descrever o comportamento da classe. A figura 5.2 apresenta a relação completa dos ícones possíveis: o primeiro ícone permite especificar a forma de recepção das mensagens, o segundo a abordagem de descrição do comportamento e o terceiro a abordagem utilizada para

troca de mensagens (maiores detalhes em relação a estes paradigmas encontram-se no capítulo 2).

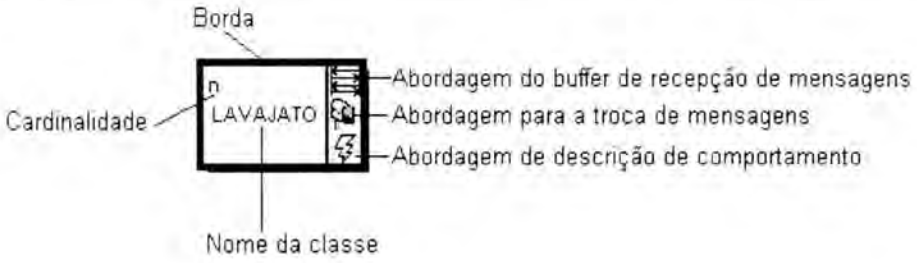


FIGURA 5.1 - Representação gráfica de uma classe

- do lado esquerdo do retângulo especifica-se, ao centro, o nome da classe e no canto superior esquerdo a cardinalidade da mesma. A cardinalidade permite definir restrições ao modelo limitando o número máximo de instâncias dessa classe no modelo (tal informação é importante para a execução do modelo).

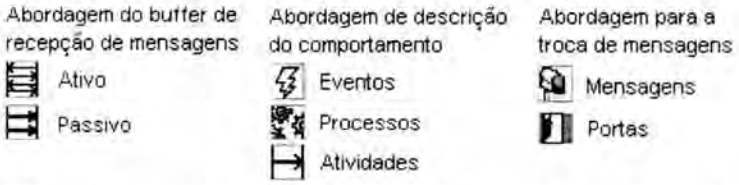


FIGURA 5.2 - Abordagens para a composição de um paradigma de simulação

Classes são detalhadas em subdiagramas onde descrevem-se as classes componentes das mesmas, caso estas existam. A identificação de um subdiagrama é feita pelo nome da classe sendo detalhada inserido em um retângulo de borda tracejada. A figura 5.3 apresenta o detalhamento de uma classe chamada LAVAJATO. Nesta pode-se observar que a classe GER_CLI possui o valor 1 para cardinalidade. Isso determina que para qualquer configuração de lava-jato que venha a ser testada só poderá haver uma instância da classe GER_CLI. No caso de não ser necessário impor restrições, indica-se "n", ou seja, sem restrições, como valor para a cardinalidade da classe.

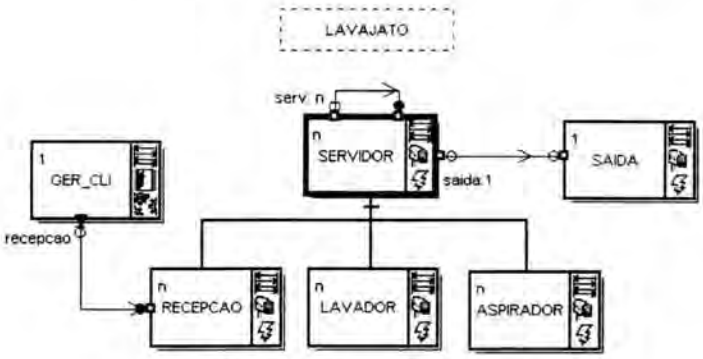


FIGURA 5.3 - Subdiagrama

Classes representadas por retângulos com bordas espessas constituem agregações, ou seja uma classe de entidade composta, cujos componentes são detalhados em um sub-diagrama. A quantidade de subdiagramas aninhados é ilimitada. Bordas simples indicam que a classe em questão não contém outros componentes. Observe na

figura 5.3 a borda espessa da classe SERVIDOR (classe composta) e a borda simples das demais (classes simples).

5.2.2 Relacionamento de herança

Herança (veja seção 1.3) define um relacionamento onde uma classe compartilha parte de sua descrição com outras. Desta forma *herança* representa uma hierarquia de abstrações onde uma subclasse herda de uma superclasse. Tipicamente, subclasses incrementam ou redefinem as estruturas estática e dinâmica de sua superclasse. Em sua versão atual, SIMOO suporta apenas herança simples. A representação gráfica deste relacionamento pode ser vista na figura 5.3. As classes SERVIDOR, RECEPCAO, LAVADOR e ASPIRADOR compõem uma hierarquia de herança na qual SERVIDOR é a superclasse.

É importante salientar que a especificação dos paradigmas de simulação na definição de uma classe é, também, uma forma implícita de se representar herança, uma vez que o conjunto de paradigmas especificado permite definição automática da especialização do elemento autônomo básico (ver capítulos 4 e 6) da qual a classe em questão será derivada. Este tipo de representação foi escolhido com o objetivo de permitir a construção de diagramas mais simples, já que todas as classes que descrevem entidades de simulação tem de ser derivadas de alguma das especializações do elemento autônomo básico.

É imposta ainda uma restrição sobre as hierarquias de herança com relação ao paradigma de simulação adotado. Uma vez que herança múltipla não é suportada, todas as classes declaradas na mesma hierarquia de herança devem, obrigatoriamente, ser definidas segundo os mesmos paradigmas.

5.2.3 Relacionamento de agregação

Agregação implica que uma entidade contém outras (veja seção 1.3). De uma maneira geral pode-se dizer que uma classe agregada "contém" outras classes-componentes, as quais "são parte da" classe agregada. Para efeitos de simulação, existe uma forma de dependência entre as entidades componentes e o agregado que os contém, uma vez que as entidades agregadas possuem o mesmo ciclo de vida da entidade agregadora. A *agregação* é vista como uma forma de encapsulamento, no qual as classes agregadoras são representadas graficamente por uma borda mais espessa que as demais.

Representando-se agregações como subdiagramas, orienta-se a construção do modelo de simulação como uma hierarquia de agregações. A principal vantagem dessa abordagem é reduzir a complexidade e aumentar a legibilidade dos diagramas. Sendo a hierarquia de agregações bem planejada, tem-se a tendência de que os subdiagramas mantenham-se simples independentemente do tamanho ou da complexidade do modelo. A figura 5.1 apresenta uma classe agregadora (borda espessa) enquanto que a figura 5.3 apresenta o subdiagrama correspondente.

5.2.4 Relacionamento de conhecimento

Relacionamentos do tipo *conhecimento* (veja seção 1.3) denotam que um objeto conhece outro (e desta forma pode se comunicar com este). Um relacionamento do tipo

conhecimento denota fraco acoplamento entre os objetos. A Figura 5.4 detalha os tipos de relacionamento de conhecimento possíveis em SIMOO.

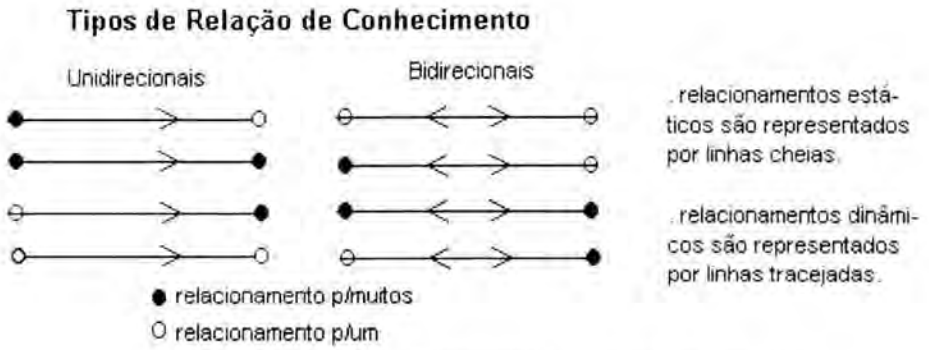


FIGURA 5.4 - Relacionamentos de conhecimento

Relações de *conhecimento* possuem sentido e cardinalidade. Quanto ao sentido podem ser uni ou bi-direcionais. Quanto à cardinalidade podem ser relacionamentos do tipo "um para um", "um para muitos" ou "muitos para muitos". A cardinalidade maior que 1 deve ser especificada. Isso pode ser feito de modo absoluto indicando-se numericamente a cardinalidade do relacionamento, ou de forma relativa utilizando-se um asterisco. O asterisco irá indicar, para o caso dos relacionamentos estáticos, que a cardinalidade é determinada pela quantidade de instâncias no diagrama de instâncias. Nos relacionamentos dinâmicos o asterisco indica cardinalidade ilimitada. Relacionamentos estáticos são aqueles que devem ser resolvidos durante a geração do modelo executável. Relacionamentos dinâmicos, por outro lado, só podem ser resolvidos em tempo de execução do modelo. Durante a geração de código pode-se determinar apenas as variáveis que irão armazenar as futuras referências.

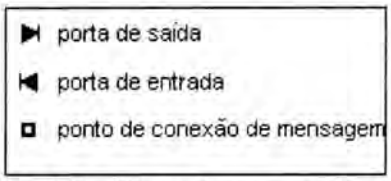


FIGURA 5.5 - Tipos de pontos de conexão

Os arcos que representam relações de conhecimento não são conectados diretamente às classes no diagrama. As conexões são feitas através de *pontos de conexão* (fig. 5.5). Os pontos de conexão podem ser de dois tipos dependendo da abordagem de comunicação utilizada. Se a abordagem escolhida for *comunicação por mensagens*, então serão usados pontos de conexão de mensagens. Se a opção for comunicação por portas, os pontos de conexão representarão portas de entrada ou saída.

Os dois tipos de pontos de conexão procuram refletir o comportamento esperado em relação à abordagem que representam. No caso da abordagem por mensagens, o nome do ponto de conexão irá corresponder ao atributo da entidade que contém os identificadores dos objetos "conhecidos". Se o relacionamento for bi-direcional, os dois pontos serão nomeados. No caso de relacionamentos uni-direcionais, apenas o ponto de onde se origina o relacionamento será nomeado. Pontos de conexão de mensagens admitem apenas uma conexão. Entre um par de classes que se comunicam por mensagens só pode haver uma única conexão deste tipo que irá representar o protocolo

de comunicação entre as duas. Por esta conexão irão transitar diferentes tipos de mensagens.

Pontos de conexão que representam portas, por outro lado, podem ser de dois tipos: de entrada ou de saída. Pontos que representam portas de entrada são associados à rotina de tratamento correspondente, enquanto pontos que representam portas de saída são associados a uma lista de entidades para as quais os valores recebidos devem ser enviados (sendo esta lista obtenível a partir das relações entre as diferentes classes). Ligações entre portas representam um canal de comunicação por onde transita apenas um tipo de mensagem. O protocolo de comunicação de um par de classes que se comunicam por portas é dado pelo conjunto de suas portas interligadas. Todo o ponto de conexão que representa uma porta deve ter um nome. Não são admitidas conexões entre duas portas de entrada ou entre duas portas de saída. Para manter a compatibilidade entre os paradigmas, entretanto, são admitidas conexões de uma porta de saída com um ponto de conexão de mensagem (neste caso o nome da porta será usado como identificador da mensagem) e de um ponto de conexão de mensagem com uma porta de entrada (a porta tratará apenas mensagens cujo identificador for igual ao identificador da porta). Portas admitem múltiplas conexões e um mesmo par de classes pode ter várias conexões entre suas portas. Na figura 5.3 pode-se observar uma porta de saída na classe GERA_CLI e pontos de mensagem nas demais.

5.2.5 Relacionamentos de criação

Relacionamentos do tipo *criação* representam a relação entre geradores de elementos da classe "X" e a própria classe "X". Em modelos de simulação é comum a existência de uma entidade permanente que seja responsável por criar instâncias de entidades temporárias segundo uma distribuição de probabilidade. No exemplo da figura 5.6, uma entidade da classe GERADOR gera entidades (instâncias) da classe CLIENTE. Classes criadas a partir do relacionamento de criação não são consideradas, obrigatoriamente, componentes da classe em cujo subdiagrama a criação está representada. Assume-se que as classes criadas a partir de um relacionamento de criação são agregadas ao modelo.

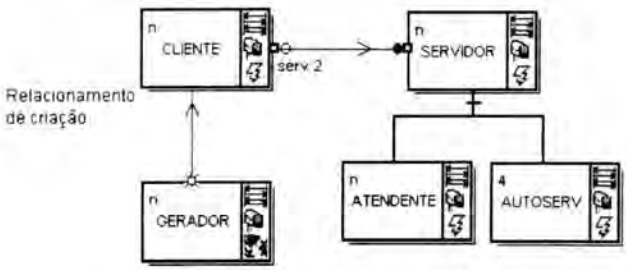


FIGURA 5.6 - Relacionamento de criação (GERADOR/CLIENTE)

5.2.6 Classes referenciadas

Declarações de classes são visíveis graficamente apenas no contexto onde são declaradas, porém, a priori, qualquer classe pode ser referenciada (reusada) em diferentes subdiagramas. Imagine, por exemplo, o modelo da arquitetura de um computador onde uma classe abstraindo a idéia de registrador pode ser utilizada tanto na descrição da unidade de controle como da memória ou da unidade lógica e aritmética. Nestas situações, para evitar que a classe tenha de ser duplicada nos diferentes contextos

correndo o risco de gerar inconsistências, cria-se uma espécie de meta-nível, no qual estas classes são representadas. Uma vez neste nível, *referências* para esta classe podem ser utilizadas em qualquer sub-diagrama e serão consideradas como se a mesma tivesse sido definida no próprio. Qualquer alteração sobre uma referência reflete-se imediatamente em todos os subdiagramas onde referências para a mesma classe são utilizadas. Alternando uso de classes referenciadas e classes comuns, é possível encapsular as declarações de classe que têm seu uso restrito a um subdiagrama e liberar a visibilidade das demais. A figura 5.7 mostra a representação gráfica de uma *classe referenciada*.

FIGURA 5.7 - Representação de uma *Classe Referenciada*

5.3 O Diagrama de instâncias

5.3.1 Justificativa da necessidade do diagrama de instâncias

Apesar de apresentar algumas particularidades relacionadas a seu uso no contexto de simulação, a idéia básica do diagrama de classes usado é bastante similar à daqueles utilizados na grande maioria dos métodos orientados a objetos existentes. Entretanto, uma característica particular de SIMOO é exigir que o usuário também crie um diagrama de instâncias, a partir do qual será gerado o modelo executável de simulação. A fim de facilitar a compreensão dos motivos que tornam interessante e necessária a criação do diagrama de instâncias, sua funcionalidade será exemplificada através do desenvolvimento do modelo do LAVAJATO já introduzido neste capítulo. Neste exemplo, a partir de um único diagrama de classes, válido para toda uma categoria de estabelecimentos de lavagem de veículos, diferentes estabelecimentos podem ser modelados.

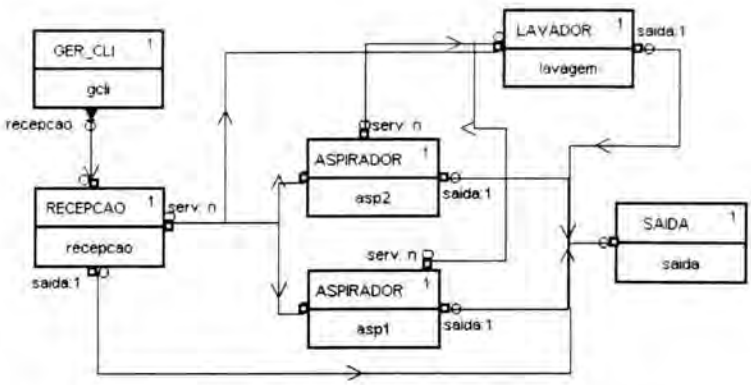


FIGURA 5.8 - Diagrama de Instancias do estabelecimento de lavagem

A figura 5.3 apresenta o diagrama de classes do LAVAJATO. Sem muitos conhecimentos sobre estabelecimentos desse tipo, é possível perceber que este diagrama descreve todo um conjunto de estabelecimentos semelhantes. Para que seja possível simular o modelo, entretanto, é necessário que se defina um estabelecimento de lavagem em particular. Isso é feito através do diagrama de instâncias. Neste, as entidades de um

modelo específico são listadas e suas relações particularizadas. A figura 5.8 apresenta o diagrama de instâncias de um estabelecimento de lavagem de veículos específico, enquanto que a figura 5.9 apresenta uma variação do mesmo. Observe que embora o diagrama de classes seja o mesmo para os dois exemplos, o exemplo da figura 5.8 apresenta dois aspiradores e um lavador, enquanto que o da figura 5.9 utiliza apenas três lavadores e nenhum aspirador.

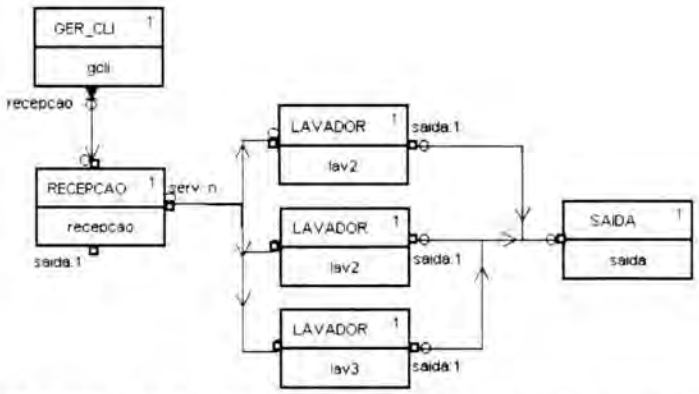


FIGURA 5.9 - Diagrama de instanciamento alternativo

Outro aspecto que reforça a necessidade do diagrama de instâncias é o uso de parâmetros de instanciamento. Em geral, conjuntos de classes que possuem elementos em comum, porém divergem em alguns aspectos, são descritos em hierarquias de herança como é o caso da hierarquia encabeçada pela classe SERVIDOR da figura 5.3. Em certos casos, porém, as diferenças não justificam tal estrutura, sendo mais facilmente representadas por parâmetros de instanciamento. O consumo de energia do ASPIRADOR ou tempo médio de lavagem do LAVADOR são exemplos.

Finalmente, existem situações onde o diagrama de classes pode resultar em diferentes representações em termos de instanciamento. A figura 5.10a apresenta um diagrama de classes onde a classe C1 possui um relacionamento do tipo *um-para-muitos* de cardinalidade 2 com a classe C2. As figuras 5.10b e 5.10c apresentam representações possíveis para este diagrama de classes em termos de instancia.

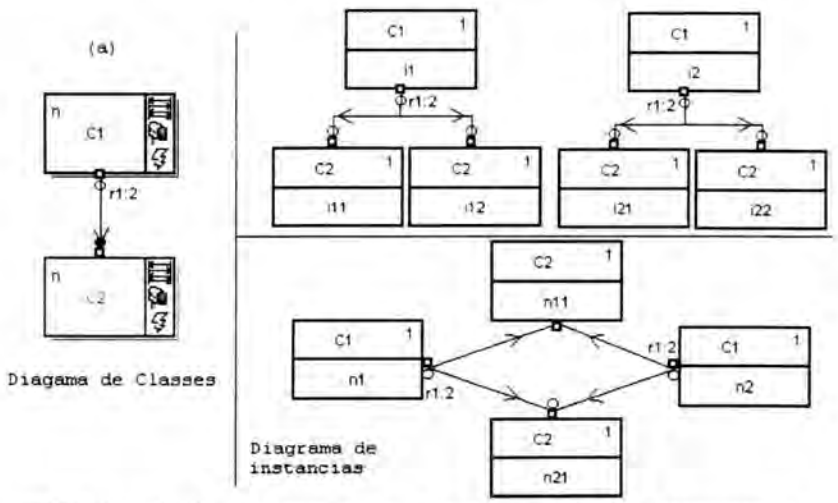


FIGURA 5.10 - Diagrama de classes com dupla interpretação

A necessidade do diagrama de instâncias leva também à conclusão de que somente se dispõe da representação completa de uma entidade de simulação após o instanciamento das classes, uma vez que as entidades de simulação são representadas por objetos e não por classes.

5.3.2 Elementos do diagrama de instâncias

Justificada a necessidade do diagrama de instâncias, pode-se analisar seus elementos. No diagrama de instâncias, os objetos são representados por caixas divididas ao meio no sentido horizontal. Na parte superior da caixa apresenta-se o nome da classe e na parte inferior o nome da variável que contém a referência para a instância correspondente. Esta variável corresponde a um atributo da classe agregadora. O número no canto superior direito indica quantos objetos estão sendo referenciados pela mesma variável.

As relações de conhecimento estáticas devem ser explicitadas no diagrama de instanciamento. O modelo do LAVAJATO (figs. 5.8 e 5.9) utiliza apenas relações estáticas. Cada ponto de conexão admite vários relacionamentos (independente do fato de ser ponto de conexão de mensagens ou portas), desde que sejam respeitados os limites de cardinalidade especificados no diagrama de classes. Pontos que não forem conectados serão considerados como relações nulas. Relações dinâmicas não são explicitadas no diagrama de instâncias. Cabe ao usuário providenciar as conexões devidas no código que descreve o comportamento das entidades.

Finalmente, conjuntos de instâncias de uma mesma classe conectadas da mesma forma, como é o caso do exemplo da figura 5.9, podem ser representados de forma alternativa (fig. 5.11).Observe que todas as instâncias de LAVADOR estão conectadas da mesma forma. A representação alternativa permite que todas as instâncias sejam referenciadas através de uma única variável e acessadas individualmente através de índices.

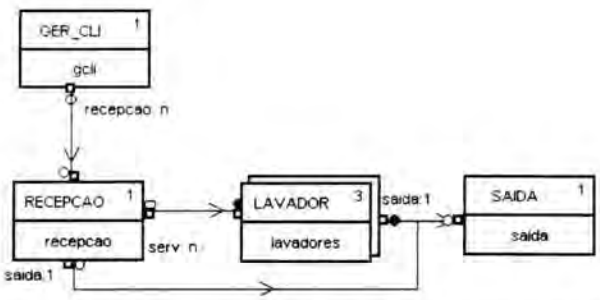


FIGURA 5.11 - Representação compacta para conjuntos de instâncias da mesma classe com as mesmas conexões

6 Implementação

O objetivo deste capítulo é descrever o protótipo construído com a finalidade de validar as idéias propostas neste trabalho. O protótipo é constituído por duas ferramentas intimamente relacionadas: a ferramenta de edição de modelos e a biblioteca de classes para simulação.

Este capítulo está dividido em duas partes. A primeira parte descreve as caraterísticas do editor de modelos e a segunda a biblioteca de classes. Como os detalhes de implementação do editor de modelos não são relevantes no contexto deste trabalho, sua descrição limita-se aos recursos disponíveis para o usuário. A descrição da biblioteca, entretanto, compreenderá tanto recursos disponíveis para o usuário, como detalhes relevantes de implementação.

6.1 A ferramenta de edição de modelos

6.1.1 Introdução

MET (Model Editor Tool) é uma ferramenta de edição de modelos de simulação por computador que utiliza dos diagramas descritos no capítulo 5 para descrever graficamente a estrutura estática do modelo e implementação dos métodos para descrever o comportamento das entidades que o compõem.

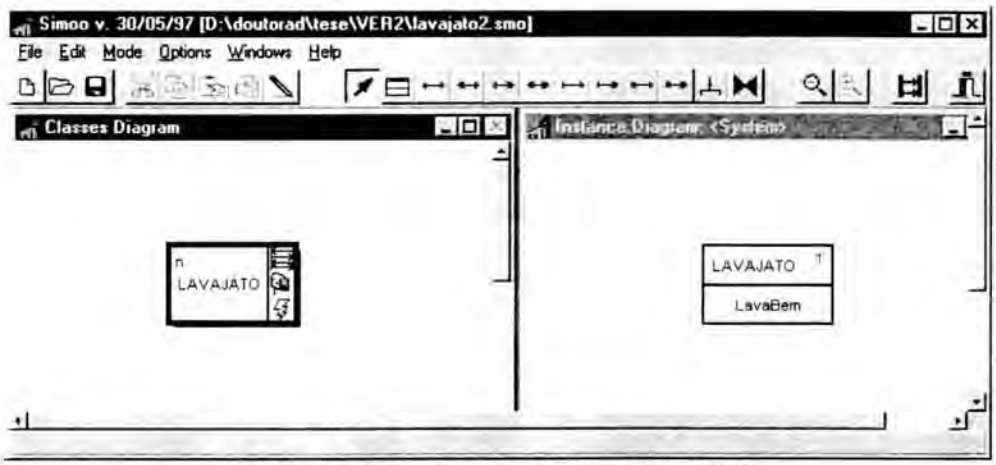


FIGURA 6.1 - Aparência do editor MET

Como descrito no capítulo 5, a descrição da estrutura estática compreende um diagrama de classes e um diagrama de instâncias. O diagrama de classes descreve as diferentes classes e por conseqüência os diferentes tipos de entidades que irão compor o modelo e seus relacionamentos possíveis. Todas as entidades do modelo são mapeadas para instâncias das classes descritas no diagrama de classes, logo, o diagrama de instanciamento permite especificar quais as entidades que compõem o modelo e quais os relacionamentos específicos entre elas. Um mesmo diagrama de classes pode servir para a construção de diferentes diagramas de instâncias.

Todas as classes definidas no diagrama de classes são derivadas de uma das classes da biblioteca de classes de SIMOO. Logo, o código C++ que descreve o

comportamento de uma entidade se utiliza dos recursos implementados pelas classes da biblioteca.

A figura 6.1 apresenta a tela básica do editor. Na parte superior da janela aparece a indicação do arquivo sendo editado. Logo abaixo seguem-se os cardápios e uma barra de ícones que facilita o acesso às operações mais comuns. Por fim, existem duas áreas de edição onde são trabalhados os diagramas de classes e instâncias respectivamente.

A fim de facilitar a apresentação dos principais recursos do editor MET, estes serão explicados através de um exemplo. Este consiste na modelagem de um estabelecimento de lavagem de veículos (um lava rápido), que deve ser simulado com o objetivo de se definir a melhor configuração para uma certa demanda de trabalho. O estabelecimento possui guichês de atendimento, onde o cliente escolhe e paga pelos serviços desejados, boxes com aspiradores de pó e boxes de lavagem de veículos. Um cliente pode optar por uma lavagem simples, lavagem completa (incluindo a limpeza interna do veículo com aspiradores) ou somente pelo uso dos aspiradores. Diagramas deste modelo foram usados nas figuras do capítulo 5.

A seção 6.1.2 apresenta a construção do diagrama de classes do modelo, a seção 6.1.3 demonstra a construção do diagrama de instâncias, a seção 6.1.4 mostra a edição do código que descreve o comportamento das entidades e, finalmente, a seção 6.1.5 trata da geração do modelo executável.

6.1.2 Construção do diagrama de instâncias

A edição de um modelo com a ferramenta MET é iniciada invariavelmente pelo diagrama de classes correspondente. Na figura 6.1 pode-se observar o primeiro nível do diagrama de classes do estabelecimento de lavagem de veículos onde é definida a classe LAVAJATO. A classe LAVAJATO representa o modelo inteiro, ou seja, inclui todas as demais. Para executar o modelo basta criar uma ou mais instâncias dessa classe.

Para criar uma classe basta selecionar o ícone correspondente (figura 6.2) e apontar a posição desejada sobre a janela referente ao diagrama de instancias. Para cada nova classe é necessário informar o nome da classe e as abordagens que compõem o paradigma de descrição. Para a cardinalidade será assumido o valor "n" (veja capítulo 5). Após a entrada destas informações a representação gráfica correspondente à classe recém criada é inserida no diagrama. A qualquer momento é possível alterar tanto a posição da representação no diagrama como suas características.

FIGURA 6.2 - Significado dos ícones

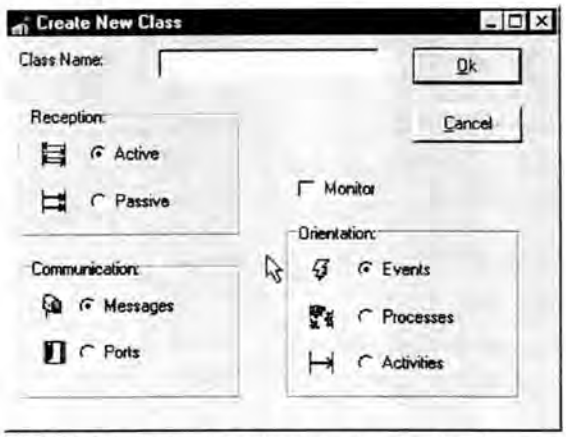


FIGURA 6.3 - Janela de criação de classes

A figura 6.3 apresenta a janela onde são informadas as características da classe no momento de sua criação. A figura 6.4 apresenta a janela que permite alterar algumas dessas características posteriormente. Note-se que além de permitir a alteração no nome da classe e sua cardinalidade, esta janela permite ainda que sejam definidos parâmetros de instanciamento para a mesma. Além disso a janela permite visualizar a relação dos pontos de conexão associados à classe em questão e as mensagens que compõem sua interface. Estas duas relações referem-se tanto aos elementos definidos na própria classe como aos elementos herdados.

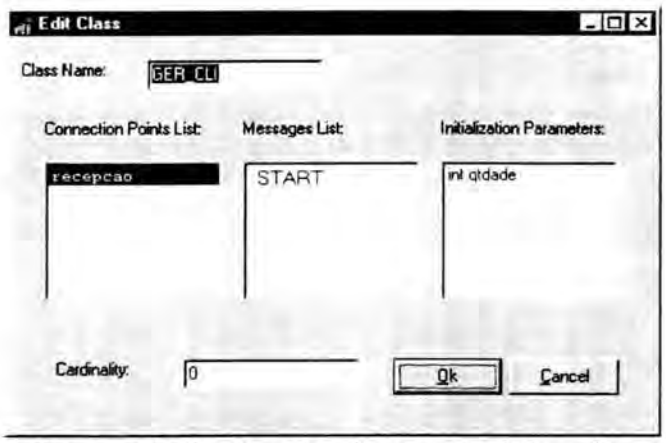


FIGURA 6.4 - Janela de alteração das características de uma classe

Como um modelo SIMOO é composto por uma hierarquia de classes, a classe que corresponde ao primeiro nível é quase sempre composta (exceto nos casos onde o modelo é muito simples e é composto apenas por uma única instância de uma única classe). Os ícones de caminhamento pelos subdiagramas (fig. 6.2) permitem navegar pela hierarquia de agregações de um modelo. Se for solicitado o subdiagrama de um componente que não possui subdiagrama, um subdiagrama vazio é criado automaticamente.

Em qualquer subdiagrama, que não o de mais alto nível da hierarquia, podem ser criadas quantas classes forem necessárias. O tamanho da área de edição pode ser regulado no cardápio "opções". A criação dos relacionamentos é feita utilizando-se os ícones adequados (fig. 6.2). A figura 6.5 apresenta o diagrama de classes do estabelecimento de lavagem de veículos.

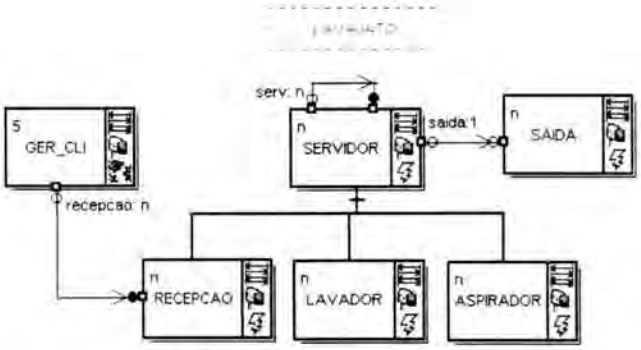


FIGURA 6.5 - Diagrama de classes

6.1.3 Construção do diagrama de instanciamento

Para cada nível do diagrama de classes deve haver um diagrama de instâncias correspondente. À medida que se navega pelos subdiagramas do diagrama de classes os diagramas de instâncias são visualizados na janela correspondente. Sem o devido instanciamento de cada um dos subdiagramas de classes, MET não é capaz de gerar um programa executável.

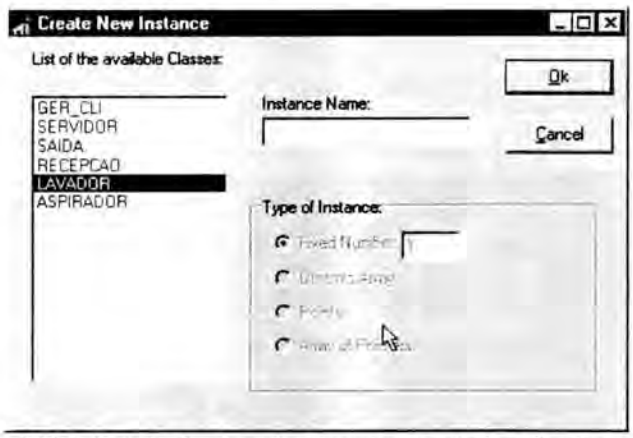


FIGURA 6.6 - Janela de criação de instâncias

A criação de uma instância é feita selecionando-se o mesmo ícone utilizado para a criação das classes, porém apontando-se sobre a área de edição das instâncias. A janela de criação de instâncias (fig. 6.6) permite selecionar a classe e indicar o identificador da instância. A lista de classes apresentada refere-se apenas àquelas que podem ser instanciadas neste nível, ou seja, a lista de classes definidas ou referenciadas no diagrama de classes correspondente. Além disso esta janela permite a criação de conjuntos de instâncias sob mesmo nome que serão identificadas por índices (veja seção 5.3.2).

Quando se cria uma instância nem todos os pontos de conexão definidos na classe correspondente precisam ser conectados. Como visto no capítulo 5, se um dos pontos de conexão não for conectado, MET assume que esta é uma relação nula no que diz respeito à instância em questão. Pontos de conexão que não serão conectados a outras instâncias não precisam ser exibidos no diagrama de instâncias. Por esta razão, a

janela apresentada na figura 6.7 acessível pelo cardápio "edição", permite controlar a visibilidade dos pontos de conexão.

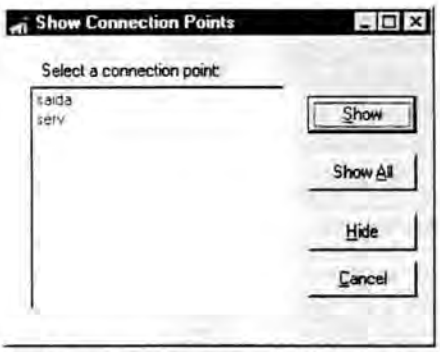


FIGURA 6.7 - Janela de controle de visibilidade dos pontos de conexão

Finalmente, permite-se também a alteração do identificador das instâncias, bem como fornecer valores para os parâmetros de instanciamento (vide Fig. 6.8)..

FIGURA 6.8 - Janela de alteração das características de uma instância

6.1.4 Edição de comportamento

Como o comportamento é único para entidades do mesmo tipo, para cada classe do diagrama de classes é possível abrir uma janela para a descrição do comportamento daquela categoria de entidades.

A janela de edição de comportamento permite que o usuário descreva a interface da classe, ou seja, o conjunto de mensagens que ela é capaz de responder bem como o código que deve ser acionado para o tratamento dessas mensagens. O código deve ser escrito em C++ considerando-se os recursos e as limitações impostas pela biblioteca. Rotinas auxiliares podem ser definidas.

A figura 6.9 apresenta a janela de edição de comportamento de uma entidade. Esta janela pode ser acionada a partir do diagrama de classes. Para cada classe pode-se definir um comportamento específico.

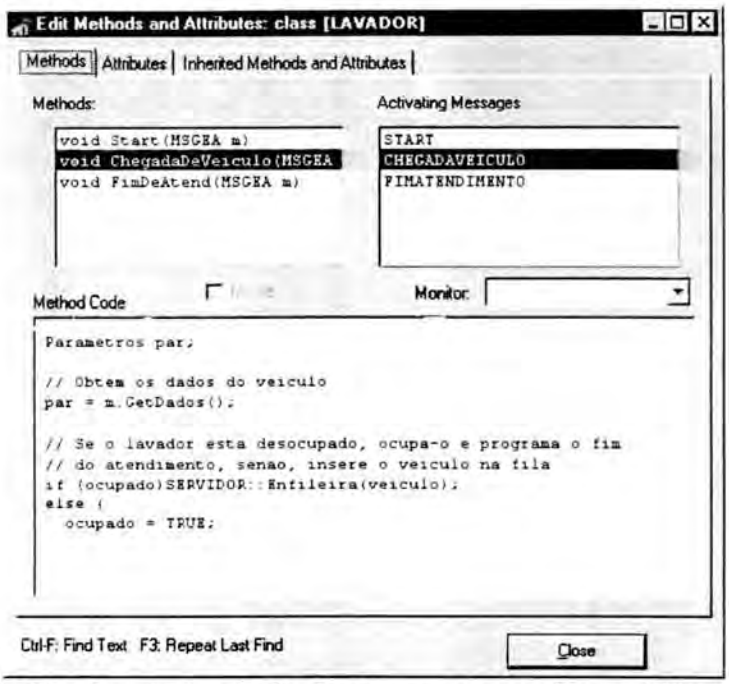


FIGURA 6.9 - Janela de edição de comportamento/Formulário de edição

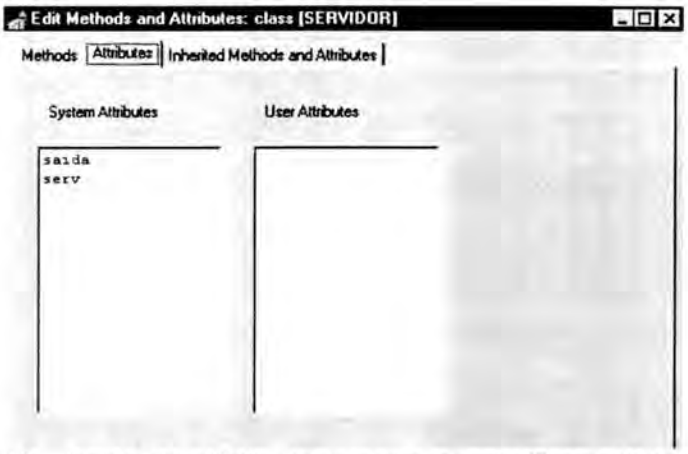


FIGURA 6.10 - Janela de edição de comportamento/Formulário de atributos

A janela de edição de comportamento é subdividida em 3 formulários: "Edição", "Atributos" e "Atributos herdados".

O formulário de edição (fig. 6.9) permite que se defina o conjunto de mensagens que compõem a interface da classe. Na parte superior definem-se as mensagens e as rotinas que devem ser ativadas em resposta. Rotinas que não são ativadas como resposta a uma mensagem recebem o valor NULL no campo correspondente. O editor dispõe de diálogos específicos para a definição das rotinas que são ativadas em resposta às mensagens que permitem detalhar seus parâmetros. Essas informações são usadas no modelo para fazer diferentes tipos de verificação em tempo de execução. Na parte inferior edita-se o código C++ correspondente à rotina/mensagem selecionada nas janelas superiores.

O formulário de atributos (fig. 6.10) permite, à esquerda, a visualização dos atributos da classe que são extraídos diretamente do diagrama de classes. À direita existe uma área onde atributos adicionais podem ser declarados.

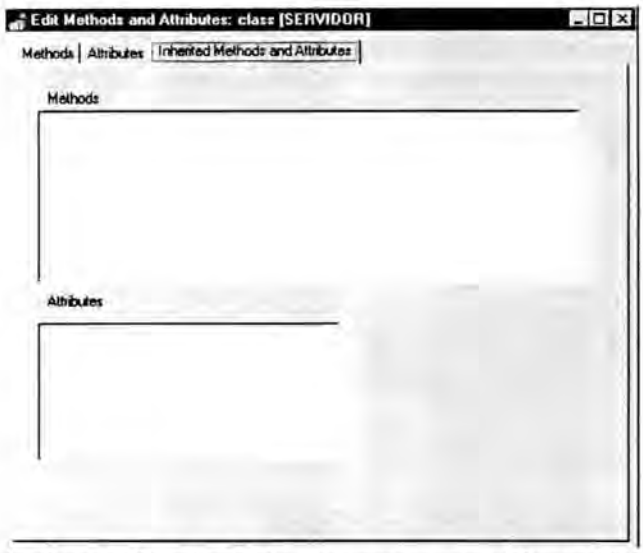


FIGURA 6.11 - Janela de edição de comportamento/Formulário de atributos

Finalmente, a janela de atributos herdados (fig. 6.11) permite apenas a visualização dos atributos herdados visíveis na classe sendo editada de maneira a simplificar sua consulta por parte do programador.

6.1.5 Geração do modelo executável

A partir da descrição do modelo, através dos diagramas e das rotinas em C++, MET é capaz de gerar um modelo executável. O primeiro passo é a geração do código fonte do modelo em C++. Em um segundo momento esse código é compilado com o compilador C++ disponível no ambiente no qual MET está inserido.

A geração do código fonte é o momento no qual MET acrescenta ao modelo descrito uma série de recursos que irão possibilitar o alto grau de interação entre o usuário e o modelo oferecido por SIMOO. Entre estes recursos pode-se destacar: a interface padrão, a meta-classe e os meta-objetos.

A interface padrão faz parte da biblioteca SIMOO que é ligada por MET ao modelo. Esta interface é composta pelo conjunto de cardápios e janelas que oferecem as opções de interação do usuário com o modelo. É a mesma para todos os modelos.

A meta-classe é definida a partir das informações do diagrama de classes. SIMOO dispõe da definição de uma meta-classe genérica que deve ser instanciada e ter suas tabelas preenchidas com as informações obtidas a partir do diagrama de classes tais como as classes disponíveis no modelo bem como a descrição de cada classe, ou seja, atributos, métodos e parâmetros dos métodos de cada uma. A meta-classe é criada junto ao gerenciador de elementos autônomos e encontra-se fortemente vinculada ao mesmo. Isso lhe permite, com algumas informações adicionais, manter o cadastro de todas as instâncias presentes no modelo bem como a capacidade para criar e destruir instâncias de elementos autônomos. MET é o responsável por inserir no código fonte do modelo as

chamadas responsáveis pelo instanciamento da meta-classe e o preenchimento de suas estruturas de dados.

Por fim, MET é responsável ainda pela criação dos meta-objetos. Para cada instância de elemento autônomo uma instância da classe meta-objeto é criada. Todas as mensagens enviadas para o elemento autônomo são interceptadas pelo meta-objeto. Este ativa suas próprias rotinas para tratar as mensagens que lhe dizem respeito ou consulta suas tabelas de maneira a ativar a rotina correspondente do elemento autônomo. Entre as mensagens que o meta-objeto é capaz de tratar encontram-se todas aquelas que se referem aos atributos (alteração ou consulta) do próprio objeto. MET é responsável não apenas por gerar o código que provê o instanciamento dos meta-objetos, mas também por preencher as estruturas de dados que armazenam as informações relativas aos atributos e métodos do elemento autônomo correspondente.

O uso de uma meta-classe e dos meta-objetos fazem com que os elementos autônomos que implementam as entidades de um modelo SIMOO possam ser questionados e ter aspectos alterados durante a execução do modelo. Enviando mensagens com o auxílio da interface padrão, o usuário pode consultar ou alterar valores de atributos de um elemento autônomo, indicar uma nova rotina para responder a uma determinada mensagem ou criar ou remover instâncias de elementos autônomos.

6.2 Protótipo da biblioteca de classes de SIMOO

6.2.1 Introdução

SIMOO é uma biblioteca de classes para simulação que utiliza o conceito de elemento autônomo, inspirado no conceito de objeto ativo descrito por Wilhelm [WIL94] (veja seção 1.6), para mapear as entidades do modelo. Um elemento autônomo é um objeto ativo, ou seja, um objeto com "thread" própria de execução cujo comportamento é descrito utilizando-se orientação a eventos e comunicação por mensagens com receptor semi-ativo (veja capítulos 2 e 3). A principal vantagem deste fato é que, a partir de um estímulo inicial representado pela mensagem START, o elemento autônomo pode entrar em um estado passivo aguardando a chegada de outras mensagens ou disparar suas próprias mensagens, visando manter um comportamento ativo. A partir da classe que implementa esse paradigma básico, são derivadas as classes que implementam as diferentes combinações de paradigmas suportadas por SIMOO. As entidades de simulação são implementadas por classes derivadas destas últimas, usando o paradigma que for mais adequado dentre os disponíveis ou derivando novos se for o caso.

A seção 6.2.2 descreve as classes básicas implementadas por SIMOO cujo conhecimento é necessário para a construção dos modelos. A seção 6.2.3 descreve os recursos disponíveis nas classes que implementam cada um dos paradigmas disponíveis e, finalmente, a seção 6.2.4 descreve alguns detalhes da implementação.

6.2.2 Classes elementares

O protótipo de SIMOO define um conjunto de classes básicas necessárias para a construção dos modelos e que por isso são dignos de menção. São elas:

- *CString*: representa um string;
- *Parametros*: armazena os parâmetros de uma mensagem;

- *MSGEA*: representa uma mensagem de elemento autônomo;
- *REFLIST*: implementa uma lista de CString;
- *RN* e *DP*: responsáveis pela geração de números aleatórios.

A classe CString oferece um conjunto de métodos para a manipulação de strings baseados nas funções tradicionais de manipulação de strings oferecidos pela linguagem C++. A vantagem de se trabalhar com CString está no fato de possibilitar a passagem de strings por valor, o que não é possível usando-se as strings tradicionais de C++ mas é imprescindível em um ambiente distribuído. Um CString é definido como um vetor de caracteres cujo tamanho máximo é definido pela constante TMAXSTRING. O valor usual dessa constante é 255.

A classe "Parametros" oferece uma maneira simples de se lidar com os parâmetros de uma mensagem. Como o número de argumentos pode variar entre os diferentes tipos de mensagens que podem ser definidos para um modelo, criou-se a classe "Parametros" para lidar com esse problema. Ela é capaz de armazenar uma lista de strings em uma estrutura que pode ser passada por valor. Apesar de trabalhar exclusivamente com o tipo CString, são oferecidos métodos que permitem manipular os valores armazenados sob a forma de valores inteiros, reais ou strings da linguagem C.

A classe MSGEA implementa o conceito de mensagem de elemento autônomo. Como se verá mais adiante, os diferentes paradigmas de simulação se utilizam de métodos com características variadas para a troca de mensagens. A mensagem em si, porém, é sempre representada por instâncias da classe MSGEA. Uma mensagem é composta por 5 elementos:

- **Identificador do emissor**: identifica o elemento autônomo que envia a mensagem.
- **Identificador do destino**: identifica o elemento autônomo para o qual se destina a mensagem.
- **Identificador da mensagem**: identifica a mensagem propriamente dita. Cada elemento autônomo é capaz de responder a um determinado conjunto de mensagens que constituem seu protocolo de comunicação. Um identificador de mensagem deve ser único no contexto de um protocolo, porém, não necessariamente no modelo.
- **Prioridade da mensagem**: a prioridade da mensagem determina a ordem em que a mesma será atendida no caso de haverem duas mensagens programadas para o mesmo tempo para o mesmo elemento autônomo. A prioridade de uma mensagem é um valor entre 0 e 255. Uma enumeração chamada "Prioridade" lista os valores mais usados (veja o guia de referência da biblioteca de SIMOO).
- **Lista de parâmetros**: toda a mensagem pode conter uma lista de argumentos representada por uma instância da classe parâmetros.

A classe REFLIST mantém uma lista de CStrings. É utilizada nos elementos autônomos para manter as listas de identificadores dos elementos com os quais eles se relacionam. Pode gerenciar listas de tamanho indeterminado ou não, conforme especificado em seus parâmetros de instanciamento.

A classe RN é a responsável pela geração de números pseudo-aleatórios usando o algoritmo TT800 [MAT94]. Possui métodos para a seleção de sementes e geração de seqüências de valores no intervalo [0,1].

A classe DP implementa a geração de números pseudo-aleatórios segundo distribuições de probabilidade. As distribuições implementadas são as descritas por Law [LAW91]. São 11 distribuições contínuas e 4 discretas. As contínuas são:

- Uniforme
- Exponencial
- m-Erlang
- Gama
- Weibull
- Normal
- LogNormal
- Beta
- Pearson Type V
- Pearson Type VI
- Triangular

Entre as discretas destacam-se:

- Bernouoilli
- Uniforme
- Binomial
- Poisson

6.2.3 Classes que implementam os paradigmas de simulação

O elemento autônomo básico é implementado pela classe EA. A classe EA é o topo da hierarquia de classes que implementa os paradigmas de simulação suportados por SIMOO (fig. 6.12). Ela implementa o chamado paradigma básico a partir do qual os demais são derivados. Este paradigma básico se utiliza de comunicação por mensagens, receptor semi-ativo e orientação à eventos.

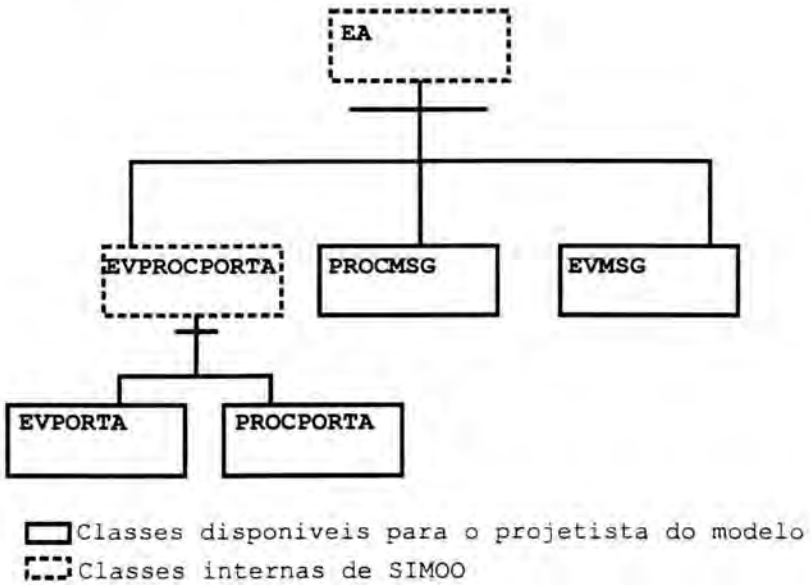


FIGURA 6.12 - Hierarquia de classes que implementa os paradigmas suportados por SIMOO

Atualmente, o usuário dispõe de 4 opções para descrever as entidades de um modelo, todas derivadas direta ou indiretamente do paradigma básico. São elas (todas utilizam receptor semi-ativo):

- Orientação a eventos e comunicação por mensagens: classe EVMSG
- Orientação a processos e comunicação por mensagens: classe PROCMSG;
- Orientação a eventos e comunicação por portas: classe EVPORTA;
- Orientação a processos e comunicação por portas: classe PROCPORTA.

Cada uma dessas classes oferece um conjunto de rotinas que o programador do modelo pode utilizar na descrição do comportamento das entidades. O modo de usar e as restrições impostas por cada conjunto de rotinas caracterizam os diferentes paradigmas. Existe um conjunto de rotinas, porém, que é comum a todos os paradigmas. Estas são implementadas pela classe EA e têm sua visibilidade assegurada ao longo da árvore de herança. São elas:

- ***CString Id(void):*** retorna o identificador do elemento autônomo;
- ***CString IdPai(void):*** retorna o identificador do elemento autônomo agregador;
- ***CString MetaClass(void):*** retorna o identificador da meta-classe;
- ***TIME Clock(void):*** retorna o tempo de simulação corrente;
- ***void Quit(void):*** solicita a destruição do elemento autônomo;
- ***int CancelEvent(TIME t,MSGEA m):*** permite o cancelamento de um evento futuro programado pelo próprio elemento autônomo. Retorna um código de erro no caso do evento a ser cancelado não existir, estiver programado para o tempo corrente ou não ter sido programado pelo EA que solicitou o cancelamento.

As classes que implementam os paradigmas que se utilizam de comunicação por mensagens, EVMSG e PROCMSG, são derivadas diretamente da classe EA.

A classe EVMSG apenas adequa a interface do paradigma básico de SIMOO implementado pela classe EA. As rotinas específicas da classe EVMSG permitem a programação de eventos futuros e consultas síncronas instantâneas, respeitando as características do paradigma de orientação a eventos com comunicação por mensagens. São elas:

- ***int Send(TIME t,MSGEA m):*** permite programar um evento (m) para um tempo futuro (t). Retorna um código de erro em caso de falha[1].
- ***int Send(MSGEA m):*** permite programar um evento (m) para o tempo corrente. Retorna um código de erro em caso de falha.
- ***int SendReceiveNow(MSGEA *m):*** permite efetuar uma consulta síncrona instantânea a outro elemento autônomo. A "thread" do elemento autônomo emissor permanece suspensa até que uma resposta ou mensagem de erro sejam enviados. A resposta deve ser fornecida, obrigatoriamente, no mesmo tempo simulado em que a consulta foi feita. Caso contrário a simulação para em estado de erro.

[1] Falhas no envio de uma mensagem pode ser relativas a destinatário inexistente ou a falta de um protocolo de comunicação estabelecido entre as classes dos elementos autônomos envolvidos. Podem ser relativas, também, ao fato do elemento receptor se encontrar ocupado. Dependendo do tipo de falha, uma mensagem esclarecedora enviada pelo gerenciador pode ser obtida com o auxílio da rotina "GetErrorMsg".

A classe PROCMSG implementa o paradigma de orientação a processos e comunicação por mensagens. Em relação a classe EVMSG as maiores diferenças estão nas rotinas "Receive" e "Wait" que permitem que o comportamento da entidade não seja descrito apenas através de rotinas instantâneas. As rotinas específicas da classe são:

- **int Send(MSGEA m):** igual à rotina equivalente de EVMSG.
- ***int SendReceive(TIME t, MSGEA *m):*** permite efetuar uma consulta síncrona. A "thread" do elemento autônomo emissor permanece suspensa até que a resposta ou uma mensagem de erro sejam enviados. A consulta pode ser programada para um tempo futuro, porém a suspensão ocorre a partir do momento da execução da rotina.
- ***int SendReceiveNow(MSGEA *m):*** igual a rotina equivalente de EVMSG.
- ***int Receive(CString mid,CString sid,MSGEA *resp):*** permite que o elemento autônomo que ativa essa função entre em estado de espera até a chegada de uma mensagem (resp) com identificador (mid) e emissor (sid) específicos. A "thread" do elemento autônomo permanece suspensa até a chegada de uma mensagem que satisfaça as condições exigidas ou de um código de erro. O símbolo "?" pode ser usado tanto no lugar do identificador da mensagem como do emissor para indicar que uma dessas condições pode ser ignorada. Seu uso em ambas não faz sentido.
- ***void Wait(TIME t):*** faz com que a "thread" do elemento autônomo que ativou esta rotina entre em estado de espera por um período de tempo simulado (indicado no parâmetro t).

A classe EVPORTA implementa o paradigma de orientação a eventos e comunicação por portas. Esta classe não é diretamente derivada de EA, e sim indiretamente a partir da classe EVPROCPORTA que implementa os aspectos relativos à comunicação por portas. As rotinas específicas deste paradigma disponíveis para o usuário são:

- **int Send(TIME t, MSGEA m):** igual à rotina equivalente de EVMSG;
- **int Send(MSGEA m):** igual à rotina equivalente de EVMSG;
- **int SendReceiveNow(MSGEA *m):** igual à rotina equivalente de EVMSG;
- **int DefinePortaIn(CString idp):** permite criar uma porta de entrada;
- **int DefinePortaOut(CString idp):** permite criar uma porta de saída;
- **int ConnectPort(CString pout, CString pdest):** permite conectar uma porta de saída com uma porta de entrada ou com um elemento autônomo que se comunica por mensagens. Esta rotina deve ser acionada pelo elemento agregador daqueles que serão conectados. Sua execução implica no envio de uma mensagem padrão ao elemento que possui a porta de saída instruindo-o a cadastrar o elemento a ser conectado em sua tabela específica. Em caso de falha a rotina retorna um código de erro[2].
- **int BreakConnection(CString pout, CString pdest):** permite desfazer a conexão entre uma porta de saída e uma porta de entrada ou com um elemento autônomo que se comunica por mensagens. A forma de funcionamento é análoga à da função "ConnectPort".

[2] Falhas na função "ConnectPort" são decorrentes da inexistência de um dos elementos a serem conectados ou de conexões inválidas como por exemplo a conexão de duas portas de entrada.

Finalmente, a classe PROCPORTA implementa o paradigma de orientação a processos e comunicação por portas. O funcionamento de suas rotinas é análogo às de mesmo nome descritas anteriormente.

- **int Send(MSGEA m):** igual à rotina equivalente de EVMSG;
- **int Receive(CString mid, CString sid, MSGEA *resp) :** igual à rotina equivalente de EVMSG;
- **int SendReceive(TIME t, MSGEA *m) :** igual à rotina equivalente de EVMSG;
- **int SendReceiveNow(MSGEA *m) :** igual à rotina equivalente de EVMSG;
- **int Wait(TIME t) :** igual à rotina equivalente de EVMSG;
- **int DefinePortaIn(CString idp) :** igual à rotina equivalente de EVPORTA;
- **int DefinePortaOut(CString idp) :** igual à rotina equivalente de EVPORTA;
- **int ConectPort(CString pout, CString pdest):** igual à rotina equivalente de EVPORTA;
- **int BreakConection(CString pout, CString pdest):** igual à rotina equivalente de EVPORTA.

6.3.4 Interação com a meta-classe

Os elementos autônomos interagem com a meta-classe por troca de mensagens. A meta-classe mantém o cadastro de todos os tipos de classes existentes no modelo bem com as informações necessárias para a criação e remoção de instâncias das mesmas. Além disso conhece a estrutura e a interface de cada classe, bem como a relação de instancias correspondente. Para facilitar a troca de mensagens com a meta-classe seu identificador é retornada pela função "MetaClass" disponível em todos os paradigmas. Entre os principais serviços oferecidos pela meta-classe que os elementos autônomos podem solicitar destacam-se:

- Criar novas instâncias;
- Remover Instâncias presentes no modelo;
- Verificar a existência de uma instância no modelo;
- Verificar a existência de uma classe no modelo;
- Verificar se a interface de uma determinada classe é capaz de reagir a uma certa mensagem.

A criação de novas instâncias é feita através do envio da mensagem "SIMOO_MO_CREATE_INSTANCE" para a meta-classe. Nos parâmetros dessa mensagem deve-se informar, na seqüência, o identificador da instância a ser criada, o identificador do elemento autônomo responsável por agregar a nova instância (não precisa ser necessariamente o elemento criador) e o identificador da classe da nova instância. Outros parâmetros específicos do tipo de classe da instância sendo criada podem ser adicionados. Para a instância criada apenas seus parâmetros específicos serão visíveis.

A remoção de uma instância é obtida com o envio da mensagem "SIMOO_MO_REMOVE_INSTANCE" para a meta-classe. Seu único parâmetro é o nome da instância a ser removida.

Para verificar a existência de uma determinada instância no modelo, envia-se a mensagem "SIMOO_MO_ISALIVE?" para a meta-classe. Esta é uma mensagem síncrona cujo parâmetro deve conter o nome da instância que se deseja verificar a presença no modelo. O retorno da mensagem é um valor inteiro: 0 se a instância não está presente no sistema e 1 caso contrário.

A verificação da existência de uma classe no modelo é semelhante a verificação da presença de uma instância. Utiliza-se a mensagem síncrona "SIMOO_MO_CLASS?" cujo parâmetro é o nome da classe e o retorno é similar ao da mensagem "SIMOO_MO_ISALIVE?".

Finalmente, para verificar se a interface de uma classe é capaz de responder a uma determinada mensagem, envia-se a mensagem síncrona "SIMOO_MO_ANSWERMSG?" cujos parâmetros são o nome da classe e o nome da mensagem e o retorno é similar ao da mensagem "SIMOO_MO_CLASS".

6.2.4 Detalhes de implementação

A especificação de um elemento autônomo básico foi vista no capítulo 4. Esta seção detalha a implementação de alguns dos aspectos especificados.

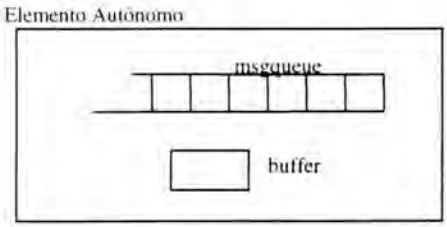


FIGURA 6.13 - Estrutura de um elemento autônomo básico

A estrutura de um elemento autônomo básico é muito simples e pode ser vista na figura 6.13. Compreende basicamente uma fila de mensagens e um "buffer" para o armazenamento das mensagens aguardadas por comandos de comunicação síncrona. A figura 6.14 apresenta a rotina básica de controle que permanece ativa durante toda sua existência.

```
void EA::WaitForMsgs(void)
  {
    MSGEA msg;

    gerea->RegistrateEA(Id(),this);//Cadastra o EA junto ao gerenciador
    while(alive)          // Enquanto o ciclo de vida do EA não se encerrar
      if (!msgqueue.Dequeue(&msg))          // Se não tem msgs p/tratar
        WaitForSingleObject(smfr_msgs,INFINITE); // Aguarda chegada de msg
      else                                          //Se tem msg p/tratar
        {
          mo->TranslateMsg(msg);//Aciona meta-objeto p/prover tratamento p/msg
          gerea->SignEventEnd();//Indica o fim do tratamento da msg corrente
        }
    gerea->CancelEARegistration(this); //Descadastra o EA do gerenciador
  }
```

FIGURA 6.14 - Rotina de espera e tratamento de mensagens de um elemento autônomo

A rotina "WaitForMsgs" controla o ciclo de vida de um elemento autônomo que consiste, basicamente, na espera e tratamento de mensagens. Inicialmente o elemento autônomo é cadastrado junto ao gerenciador de elementos autônomos. A

responsabilidade por esta operação cabe a rotina "RegisterEA" que providencia a inicialização de todos os atributos de controle do elemento autônomo e cria uma entrada correspondente na tabela do gerenciador.

Uma vez cadastrado o elemento autônomo, inicia-se o laço que gerencia sua vida ativa. Este processo consiste em verificar a presença de mensagens na fila (rotina "Dequeue" do objeto "msgqueue") e ativar a rotina "TranslateMsg" do meta-objeto para que seja providenciado o tratamento adequado à mensagem. Quando o tratamento da mensagem corrente se encerra, esse fato é comunicado ao gerenciador (rotina "SignEventEnd") e o processo é reiniciado. No caso de não haverem mensagens na fila, o elemento autônomo entra em um estado de espera controlado por um semáforo (referenciado pela variável "smfr_msgs"). Este suspende indefinidamente o processamento do elemento autônomo até ser "acordado" pela inserção de uma mensagem na fila.

O laço de espera e tratamento de mensagens permanece ativo até que seja solicitada a remoção do elemento autônomo. Isso é sinalizado na variável "alive" de maneira que o laço é interrompido. Quando isso acontece a rotina "CancelEARegistration" elimina a entrada correspondente da tabela do gerenciador e a "thread" do elemento autônomo encerra-se naturalmente com o fim da rotina "WaitForMsgs". A rotina "CancelEARegistration" preocupa-se, também em verificar a necessidade de cancelar mensagens pendentes.

As mensagens são inseridas na fila de mensagens do elemento autônomo pelo gerenciador de elementos autônomos. Um aspecto interessante da implementação de SIMOO, porém, é que o gerenciador de elementos autônomos de SIMOO não possui "thread" própria de execução. Suas rotinas são acionadas pelos próprios elementos autônomos toda a vez que se encerra o tratamento de uma mensagem através da rotina "SignEventEnd".

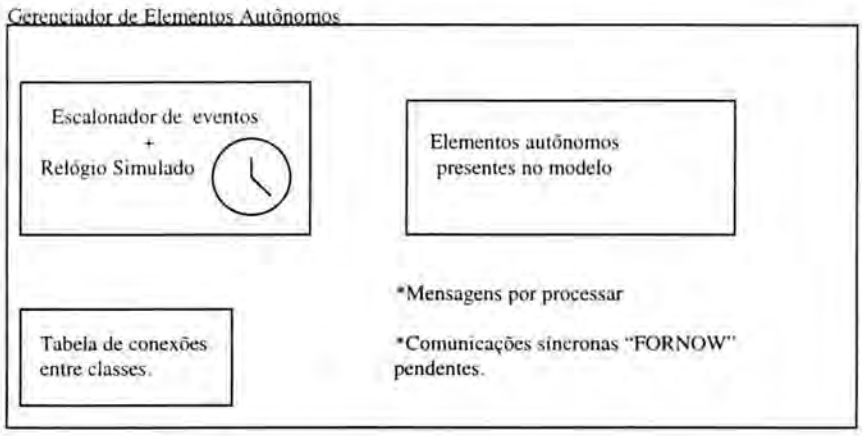


FIGURA 6.15 - Estrutura do gerenciador de elementos autônomos

A figura 6.15 apresenta a estrutura do gerenciador de elementos autônomos. Entre seus componentes pode-se destacar a tabela que mantém a lista dos elementos autônomos presentes no modelo, o escalonador de eventos e o relógio de simulação, o contador que controla a existência de mensagens sendo processadas nos elementos

autônomos, a tabela de conexões entre classes de elementos autônomos e o contador que indica se existem comunicações síncronas "FORNOW" (veja capítulo 4) pendentes.

A principal tarefa do gerenciador de elementos autônomos é controlar a troca de mensagens entre os elementos autônomos e o avanço do relógio simulado. Quando um elemento autônomo deseja enviar uma mensagem (a programação de eventos, como já foi visto, é feita pelo envio de mensagens) utiliza-se de uma das rotinas disponibilizadas pelo paradigma em uso. Esta rotina é responsável apenas pela inserção da mensagem no escalonador de eventos. Para que tal possa ocorrer, primeiramente a validade da mensagem é verificada. Os identificadores das partes envolvidas são conferidos, bem como verifica-se a existência de protocolo de comunicação entre ambos (consultando a tabela de conexões entre classes).

Toda a vez que o tratamento de uma mensagem se encerra, a rotina "SignEventEnd" é acionada. Além de indicar o fim do processamento de mais uma mensagem essa rotina verifica o escalonador de eventos e o relógio de simulação. Como durante o processamento das mensagens disparadas para o tempo corrente novas mensagens para o tempo corrente podem ser programadas, primeiramente a rotina "SignEventEnd" verifica se não existem mensagens pendentes para o tempo de simulação corrente. Se tal ocorrer, a rotina dispara todas as mensagens pendentes, inserindo-as nas filas de mensagem dos elementos autônomos a que se destinam. Encerrando esta tarefa a rotina se encerra.

Cada vez que uma mensagem é inserida na fila de um elemento autônomo, o contador de mensagens "por processar no tempo corrente" do gerenciador é incrementado. Já se viu que cada vez que o tratamento de uma mensagem se encerra esse contador é decrementado. Esse mecanismo permite que se controle o momento em que o relógio de simulação deve avançar. Se a rotina "SignEventEnd" não encontra mensagens pendentes para o tempo corrente, ela verifica a possibilidade de avançar o relógio de simulação. Se o contador de mensagens por processar não está zerado, significa que ainda existem elementos autônomos processando mensagens no tempo corrente, logo o relógio não deve avançar e a rotina se encerra.

Eventualmente a execução da rotina "SignEventEnd" por parte de algum dos elementos autônomos presentes no modelo irá encontrar o escalonador de eventos sem nenhuma mensagem pendente para o tempo corrente e o contador de mensagens por processar zerado. Neste caso, a rotina providencia o avanço do relógio para o próximo tempo para o qual existem mensagens programadas e dispara as mesmas.

A última possibilidade é aquela em que não existem mensagens pendentes no escalonador de eventos para qualquer tempo e o contador de mensagens por processar está zerado. Neste caso a simulação chegou ao fim.

O fato do gerenciador de elementos autônomos não possuir "thread" própria de execução, além de simplificar a sincronização dos elementos autônomos, evita o gasto de CPU equivalente ao controle de mais uma "thread".

A implementação atual de SIMOO prevê a execução dos modelos em um ambiente "multi-thread" baseado em um ou mais processadores, porém com memória

compartilhada, uma vez que existe uma tabela central que armazena referências para todos os elementos autônomos presentes no modelo. Essas limitações, entretanto, não se refletem no modelo uma vez que os elementos autônomos não compartilham áreas de memória ou variáveis comuns. Este fato facilita o transporte de SIMOO para um ambiente realmente distribuído visto que os modelos já desenvolvidos poderão ser recompilados sem a necessidade de adaptações. As alterações necessárias no SIMOO não devem atingir grandes extensões de código, limitando-se ao gerenciador de elementos autônomos.

7 Trabalhos Correlatos - Análise Comparativa

Este capítulo discute um conjunto de ambientes de simulação orientados a objetos descritos na literatura que, acredita-se, são uma amostra representativa do estado da arte. O objetivo é fazer uma análise comparativa entre os ambientes descritos, os objetivos propostos no capítulo 1 e o ambiente proposto, SIMOO.

As seções 7.1 a 7.13 que seguem apresentam a descrição e análise de cada ambiente. A seção 7.14 apresenta um quadro comparativo bem como algumas conclusões.

7.1 Biblioteca de classes de Shewchuck e Chang

A biblioteca de classes de Shewchuck e Chang [SHE91] é um exemplo de como pode-se aplicar o paradigma de orientação a objetos no desenvolvimento de modelos de simulação. Implementada em C++, constitui-se de um conjunto de classes organizadas em camadas, onde cada camada especializa o domínio de aplicação. O paradigma de simulação adotado utiliza a abordagem orientada a eventos, para a descrição do comportamento, e a abordagem de comunicação livre, para a troca de informações entre as entidades.

A primeira camada da biblioteca não tem seu domínio de aplicação restrito a sistemas de simulação. Ela implementa um conjunto de estruturas de dados genéricas, tais como listas dinâmicas, que são aproveitadas na implementação das camadas subseqüentes.

A segunda camada implementa classes de suporte a sistemas de simulação. Algumas, tais como as que implementam o relógio, o calendário, e o gerador de números aleatórios, são realmente genéricas. Outras, como as que implementam filas e coletores de estatísticas, foram desenvolvidas de maneira orientada ao uso na terceira camada. A maior ausência neste nível é a falta de uma classe que implemente a máquina de simulação. Embora sejam fornecidas classes que implementam o relógio e o calendário (escalonador de eventos), estas não funcionam sozinhas. A máquina de simulação, ou seja, o algoritmo de disparo dos eventos, deve ser implementado no modelo.

A terceira e última camada apresenta um conjunto de classes específicas para a construção de modelos de sistemas de manufatura. Implementa classes que representam peças, máquinas e o sistema de produção. As classes que representam as máquinas e as peças contêm as características comuns a esses elementos. Na abordagem utilizada as máquinas contêm informações tais como tempos de atendimento, intervalos entre falhas, vida útil etc. As peças armazenam a rota de seu processo de manufatura, ou seja, a seqüência de passos pelos quais devem passar para completar seu processo de produção. Um modelo, desta forma, será composto por um conjunto de máquinas e um conjunto de peças que competem pelas mesmas. Não existem objetos geradores de peças. A descrição dos diferentes tipos de peças deve ser armazenada em estruturas de dados específicas. Os eventos correspondentes a sua criação devem ser inseridos no calendário no início do programa.

A estratégia adotada revela o uso da "ótica do cliente" (veja seção 2.4.2.7). Uma característica interessante é que, embora tipos específicos de máquinas e peças possam ser derivados, as classes fornecidas não são abstratas e podem ser instanciadas diretamente na construção do modelo. Por fim, cabe destacar que tanto peças quanto máquinas agregam instâncias das classes de coleta de estatísticas, fornecidas na segunda camada, entre seus atributos. Desta forma a coleta de estatística é feita automaticamente.

7.2 PRISM

PRISM [VAU91] é um sistema de simulação de propósitos gerais orientado a objetos que prioriza um alto grau de interatividade entre o usuário e o modelo.

PRISM incentiva a construção de modelos hierárquicos. Entidades de simulação, construídas a partir da derivação da classe MODEL, são consideradas modelos completos. Instâncias da classe MODEL, porém, podem referenciar outras instâncias da mesma classe permitindo a construção hierárquica do modelo.

A máquina de simulação é implementada por uma instância da classe SIMULATOR. Cada entidade ou modelo deve possuir uma referência para esta instância de forma a poder programar ou cancelar eventos futuros.

Uma das principais vantagens de PRISM é o uso de planilhas simbólicas. Embora uma entidade possa ter seus atributos declarados em C++ da maneira usual, recomenda-se que eles sejam declarados em uma planilha simbólica. Cada entidade dispõe de uma planilha cujas células podem conter valores ou expressões em C++. Embora possua o comportamento normal de uma planilha, ou seja, a alteração de uma célula pode repercutir nas demais, a planilha não é organizada na forma de uma matriz, e sim como uma lista de células referenciadas por nomes (por isso o nome planilha simbólica). O uso da planilha permite um alto grau de interação durante a execução do modelo. Apenas com a simulação suspensa, sem a necessidade de recompilação ou reinicio da simulação, é possível não apenas alterar o valor dos atributos como remover ou inserir novos, através de operações sobre a planilha.

Os autores de PRISM defendem que, embora as classes base de PRISM utilizem uma abordagem orientada a eventos e comunicação livre, qualquer tipo de paradigma pode ser derivado. Apresentam como exemplo um conjunto de entidades para a implementação de modelos de fila de espera que se utilizam de portas para se comunicar. Citam ainda a possibilidade de se derivar a abordagem orientada a processos, porém isso se torna extremamente complexo devido à necessidade de se gerenciar co-rotinas. Embora aparentemente não existam restrições a que cada entidade ou modelo seja descrito segundo o paradigma mais adequado, a compatibilidade entre os mesmos deve ser assegurada por quem os desenvolver.

O editor de modelos de PRISM permite a criação de ícones que podem ser associados a classes. O próprio ambiente de edição pode ser usado para execução permitindo, inclusive, a remoção e inserção de instâncias durante a simulação.

Finalmente, animação em PRISM é provida de forma automática com a finalidade única de depuração do modelo. Podem ser vistas transações competindo por recursos e aguardando em filas. No caso do paradigma descrito, essa forma de animação

é indicada como interessante para a determinação de gargalos no sistema. Outras formas de animação, embora possíveis de serem implementadas, são desencorajadas.

7.3 DOSE

DOSE [MAK91], é um ambiente de simulação de propósitos gerais onde um modelo pode ser construído de maneira hierárquica usando tanto componentes definidos pelo usuário como componentes previamente definidos em uma biblioteca.

DOSE é composto por uma biblioteca de classes em C++ que é ligada ao programa do usuário de maneira a prover o componente básico, a partir do qual são derivadas as entidades de simulação, bem como outros recursos adicionais tais como a máquina de simulação e recursos para a coleta de estatísticas. DOSE não provê um ambiente de trabalho, obrigando o usuário projetista a usar o ambiente do compilador C++ disponível.

Componentes DOSE são objetos ativos caracterizados por um paradigma fixo que usa orientação a eventos e comunicação por portas. O objetivo em DOSE é prover o encapsulamento total do comportamento e dos dados de cada componente de maneira a facilitar seu reuso. Um modelo DOSE é composto por um conjunto de componentes interligados.

Componentes DOSE conhecem apenas suas portas, não conhecendo os elementos com os quais se comunicam em tempo de execução. O ambiente utiliza um gerenciador central de conexões que permite, entre outros aspectos, a alteração dinâmica das conexões durante a execução. Esta característica visa facilitar o teste de diferentes configurações bem como a construção de modelos de sistemas que têm sua configuração alterada no caso de falhas.

DOSE permite que novos componentes sejam criados por herança. Não existem recursos para facilitar a criação de bibliotecas de componentes, ficando a cargo do usuário gerenciar suas classes C++. DOSE oferece ainda, um conjunto bastante rico de classes para a coleta de estatísticas e geração de histogramas.

7.4 ModSim II

ModSim [BEL90] é uma linguagem de propósitos gerais, cuja sintaxe é inspirada na sintaxe de Modula II, a qual foram acrescentados recursos próprios para o desenvolvimento de sistemas de simulação. A portabilidade de ModSim advém do fato de não existir um compilador ModSim, e sim pré-processadores que convertem programas ModSim em programas C, permitindo que se use o compilador C disponível.

ModSim suporta a maioria das construções de orientação a objetos disponíveis na linguagem C++, entre as quais pode-se destacar herança simples ou múltipla, ligação dinâmica, polimorfismo, abstração de dados e encapsulamento. Da mesma forma que a linguagem C, ModSim favorece o desenvolvimento de programas modulares. Cada módulo pode conter um conjunto de rotinas e declarações de classes e pode ser compilado em separado.

Em relação a recursos para simulação, ModSim oferece "Ask methods" e "Tell Methods". "Ask Methods" correspondem a rotinas cuja execução é considerada

instantânea em termos de tempo de simulação. "Tell methods" correspondem a rotinas que podem conter instruções do tipo "wait", ou seja sua execução pode levar várias unidades de tempo simulado. Pela análise destes comandos conclui-se que o paradigma de simulação de ModSim utiliza orientação a processos. A troca de informações entre entidades utiliza troca de mensagens com o auxílio da palavra reservada "Ask". Esta característica permite que os modelos possam ser executados sem alterações em ambientes distribuídos. Uma versão experimental para máquinas com múltiplos processadores vem sendo desenvolvida.

Quanto a recursos para visualização de resultados, ModSim oferece uma biblioteca de elementos gráficos. Embora de utilização simples, seu controle tem de ser inserido no modelo. Não são oferecidos recursos para interação entre o usuário e o modelo.

7.5 MODES

MODES (Message Oriented Discret Event Simulation) [CHA94] é um ambiente de simulação de propósitos gerais desenvolvido em Common Lisp que visa explorar os conceitos de simulação orientada a objetos e simulação distribuída.

MODES utiliza o modelo de abstração do ator para explorar a troca de mensagens como base da computação concorrente. Todos os atores possuem um identificador único que pode ser comunicado aos demais permitindo a troca de informações. MODES permite troca de mensagens síncronas ou assíncronas. As mensagens enviadas para um ator são enfileiradas em um buffer próprio. Os atores são livres para selecionar as mensagens deste buffer ou não. O paradigma de simulação implementado pelos atores se utiliza, então, de comunicação por mensagens, orientação a eventos e receptor ativo.

MODES é um ambiente de simulação distribuída. Entidades mapeadas em atores são usadas como unidade de distribuição. MODES não possui, porém, recursos para prever erros de causalidade, exigindo que as condições para o envio de uma mensagem sejam verificadas com antecedência. O mesmo ocorre no que se refere a "dead-locks".

7.6 SIM++

SIM++ [BAE91] é um pacote de classes e rotinas em C++ especialmente projetadas para prover um ambiente determinístico de escalabilidade transparente (veja seção 1.6) para o desenvolvimento de simulações distribuídas orientadas a objetos. Os eventos recebidos por uma entidade são armazenados em um conjunto, podendo ser selecionados ou não. Serão chamados de eventos futuros aqueles que possuem tempo de evento maior ou igual ao tempo atual da entidade sendo executada e de eventos atrasados os que tem um tempo menor que o tempo de simulação corrente. SIM++ possui um conjunto de primitivas de simulação que permite manipular o conjunto de eventos de uma entidade. Quando uma entidade seleciona um evento usando uma destas primitivas, especifica um conjunto de condições definidas pela aplicação usando um valor conhecido como predicado. Um evento é selecionado se satisfaz um determinado predicado.

SIM++ foi desenvolvido objetivando explorar um maior grau de paralelismo deixando em segundo plano questões tais como interação ou visualização. No tocante à

simulação distribuída, SIM++ adota uma estratégia não conservativa para o disparo dos eventos que oferece à entidade maior liberdade na escolha das mensagens a serem processadas.

O atendimento às mensagens por parte de cada entidade é seqüencial, porém a entidade é livre para atender a mensagem que for mais adequada para seus objetivos. O tempo simulado é individual para cada entidade, que pode avançá-lo explicitamente ou quando trata uma mensagem cuja informação temporal indica um tempo maior que o corrente. As mensagens que permanecerem na fila com informação temporal com valor menor que o tempo corrente da entidade passam a ser classificadas como atrasadas e o modelo deve saber como lidar com as mesmas. Esta estratégia deixa para o modelo a tarefa de tratar eventuais erros de causalidade.

7.7 VISE

O sistema VISE [LIN95] é um ambiente de simulação composto por um conjunto integrado de ferramentas que oferece recursos para a atividade de simulação de uma maneira visual e interativa. Embora não utilize conceitos de orientação a objetos, foi incluído nesta análise por seu potencial de interação entre o usuário e o modelo.

O ambiente utiliza a linguagem SIMSCRIPT como ferramenta de modelagem e coloca à disposição do usuário um conjunto de recursos de interação, controle e visualização. Os recursos de controle permitem que a execução seja suspensa, reativada, que se tenha acesso à lista de eventos ou condições de parada. Pode-se especificar ainda a execução por um determinado número de unidades de tempo ou até a ocorrência de um determinado evento. Os recursos de visualização permitem que se especifique associações entre variáveis e ícones a fim de que se possa visualizar de maneira gráfica o andamento da simulação. Durante a simulação é possível alterar os parâmetros dessas associações.

7.8 A biblioteca de classes de Kocher e Lang

A biblioteca de classes de Kocher e Lang [KOC94] é uma biblioteca de classes em C++ que visa o desenvolvimento de modelos hierárquicos complexos. Um modelo é visto como um conjunto de componentes interligados por portas. Cada componente pode ser refinado quantas vezes forem necessárias. O paradigma empregado usa orientação a eventos e comunicação por portas. Não existe a possibilidade de se alterar esse paradigma básico.

O tratamento das mensagens por parte de uma entidade é feito através do instanciamento de "tratadores de mensagens". Uma entidade pode ter um ou mais tratadores de mensagens que podem tratá-las diretamente ou repassá-las para os níveis inferiores da hierarquia. A facilidade com que os "tratadores de mensagens" podem ser substituídos facilita o teste das diferentes hipóteses. Eventuais filas de mensagens devem ser implementadas por estes.

As possibilidades de controle da simulação nesta biblioteca enfatizam o controle relacionado à coleta de estatísticas. Indicação de períodos de "Warm-up" [LAW91], programação de diversas reproduções independentes [GOG95], bem como o cálculo de médias e variâncias são previstos na interface oferecida.

Finalmente, esta biblioteca oferece recurso para impressão de resultados. Estes podem ser particularizados para cada modelo através do uso de arquivos de "estilo".

7.9 EXSIM

EXSIM [ZHE93] é um ambiente de simulação de propósitos gerais que se utiliza de um editor gráfico para desenvolver modelos compostos por componentes conectados por portas.

As abstrações básicas de um modelo em EXSIM são nodos, portas e arcos, as mesmas de PRISM (seção 7.2), da biblioteca de Kocher e Lang (seção 7.8) e de alguns dos paradigmas de SIMOO e são oferecidas em uma biblioteca de classes em C++. Para desenvolver seus próprios modelos, o usuário deve derivar suas próprias classes a partir das abstrações básicas. Um recurso interessante oferecido por EXSIM é o editor de modelos. Este permite que o usuário associe ícones às classes especializadas a partir da biblioteca básica. Desta forma, se as classes necessárias já foram desenvolvidas, a construção do modelo irá corresponder à simples conexão de ícones em um editor gráfico. Esta abordagem favorece o reuso de classes. O editor oferece também a possibilidade de se associar parâmetros de instanciamento às classes desenvolvidas. Os valores dos mesmos são questionados ao usuário no momento que este insere o ícone no diagrama. Existe ainda a opção de geração do modelo executável a partir do diagrama.

EXSIM não oferece nenhum recurso para animação do modelo ou visualização de resultados, nem mesmo uma interface de controle do andamento da simulação.

7.10 A Biblioteca de Classes de Abell e Judd

A biblioteca de classes de Abell e Judd [ABE93] também se utiliza de um modelo de componentes interligados por portas semelhante aos descritos nas seções 7.2, 7.8 e 7.9. Vale a pena destacar, entretanto, a maneira como a estrutura hierárquica do modelo é implementada utilizando a idéia de "objetos de grupo".

Em princípio todo a instância descrita pela especialização de uma das classes fornecidas é considerada um elemento atômico. Agregações devem ser construídas com auxílio da classe "Group Object" que tem condições de gerenciar estruturas desse tipo. As principais vantagens de seu uso estão relacionadas com as facilidades em se alterar a hierarquia em tempo de execução.

7.11 VMSS

VMSS [KAM96] é um sistema de simulação de sistemas flexíveis de manufatura especialmente orientado para o controle de veículos de controle automático (conhecidos pela sigla AGV em inglês) que integra dois grandes conceitos: modelagem interativa visual e simulação interativa visual.

VMSS é composto por três grandes módulos: Escritor, Teatro e Diretor. O módulo "Escritor" é aquele onde o modelo é efetivamente construído. Um editor gráfico permite que diferentes tipos de equipamentos sejam escolhidos em uma palheta e dispostos sobre a área de trabalho. Cada tipo de equipamento possui um ícone associado e é descrito por uma classe com parâmetros de instanciamento que devem ser informados pelo usuário através do preenchimento de tabelas.

O "Teatro" oferece o palco onde as simulações serão executadas. Sua grande vantagem é a facilidade como mantém os resultados das diferentes execuções do modelo e suas variadas configurações. Animações são providas automaticamente visto que o conjunto de elementos que pode ser instanciado é fixo e de comportamento conhecido. O estado da simulação e de cada componente são permanentemente exibidos.

Finalmente o "Diretor" é o módulo de controle que permite o funcionamento integrado do "Escritor" e do "Teatro".

Os pontos fortes de VMSS são o alto grau de interação entre o usuário e o modelo e as facilidades para armazenar e gerenciar os resultados oriundos de diferentes configurações. Outra característica interessante é a existência de um "assistente inteligente". Este avisa sobre operações incorretas e sobre os próximos passos.

7.12 SMOOCHES

SMOOCHES [BAR96] é um ambiente de simulação de propósitos gerais que permite a descrição de modelos hierárquicos orientados a objetos através de máquinas de estados.

Entidades em SMOOCHES são mapeadas para elementos autônomos que empregam um paradigma que se utiliza de orientação a eventos e comunicação por mensagens. O sistema de comunicação por mensagens é ligeiramente diferente do adotado por SIMOO na medida em que admite "broadcasts" seletivos ou não (SIMOO não admite qualquer tipo de "broadcast" quando se utiliza de comunicação por mensagens).

Para cada tipo de entidade pode-se associar um ícone e definir um diagrama de estados que descreve seu comportamento. O uso dos diagramas de estado revela uma preocupação com o aspecto de descrição formal do modelo e a conseqüente simplificação da fase de validação do mesmo.

SMOOCHES oferece ainda alguns tipos de elementos monitores que podem acompanhar tanto as trocas de mensagens entre as entidades, assim como o suceder dos estados de um determinado diagrama de estados. Recursos para tratar restrições características de sistemas de tempo real bem como para desenvolver modelos que podem executar em ambientes distribuídos são providos.

Finalmente, SMOOCHES não se preocupa com a interação entre o usuário e o modelo ou com recursos específicos para visualização de resultados.

7.13 VSE

VSE (Visual Simulation Environment) [BAL97] é um ambiente de propósitos gerais que permite o desenvolvimento de modelos de simulação discreta orientados a eventos utilizando-se do paradigma de orientação a objetos na construção de modelos hierárquicos aos quais podem ser associadas representações visuais aos diferentes níveis da hierarquia.

VSE incorpora uma biblioteca de classes chamada VSLlibrary. Esta implementa as classes a partir das quais todos os componentes do modelo são derivados. Os modelos

são desenvolvidos em um editor interativo onde, usando um linguagem textual específica e orientado por uma série de janelas, o usuário especifica classes de objetos, hierarquias de herança bem como a hierarquia de agregações que compõem o modelo. O comportamento das entidades é descrito por orientação a eventos e troca de mensagens.

Modelos VSE podem conter componentes estáticos e componentes dinâmicos. Componentes estáticos permanecem parados durante toda a simulação enquanto que componentes dinâmicos podem mover-se dentro do modelo.

Cada componente estático pode ser detalhado em subcomponentes. Componentes que não são folhas da hierarquia podem ter uma representação visual associada. É possível sensibilizar pontos da representação visual de um determinado nível da hierarquia de modo que quando indicados pelo usuário permitem a abertura de janelas que permitem visualizar a representação visual associada com os níveis inferiores.

Componentes dinâmicos são organizados da mesma forma que os componentes estáticos. A diferença básica reside nas possibilidades de mover-se e "entrar" dentro de outros componentes estáticos ou dinâmicos. Se um componente dinâmico entra em outro componente dinâmico e este último se move, o movimento aplicado ao componente mais externo reflete-se nos mais internos. Como por exemplo pode-se citar o modelo de um avião onde os passageiros (componentes dinâmicos) entram no avião (outro componente dinâmico). Se o avião se mover, os passageiros se movem junto, embora possam continuar deslocando-se dentro do avião.

Durante a simulação o usuário pode abrir e fechar janelas de maneira a poder acompanhar a animação de todos os aspectos do modelo. Em uma janela de texto, resultados parciais resultantes de comandos específicos inseridos no modelo podem ser visualizados. Ao final da simulação arquivos de resultados são gerados e podem ser analisados com auxílio de uma ferramenta específica. São oferecidos recursos para o cálculo de médias, intervalos de confiança etc. Recursos para tratamento dos dados de entrada, determinação de distribuições de probabilidade entre outros também são oferecidos.

7.14 Análise comparativa

A tabela 7.1 permite comparar as diferentes características analisadas em cada um dos ambientes estudados. Note que, de forma coerente com o restante do trabalho, os ítens analisados referem-se, em sua grande maioria, a aspectos de modelagem, visualização e execução distribuída.

Diversos ambientes, entre os quais pode-se destacar PRISM, EXSIM, VMSS e SMOOCHES, oferecem a possibilidade de edição gráfica do modelo. Em todos, porém, o usuário associa uma representação gráfica à descrição de uma entidade ou componente e, a partir dessas representações permitem que se descreva a estrutura do modelo como uma rede de ícones. SIMOO se utiliza de uma abordagem diferenciada. São utilizados diagramas que permitem a descrição da estrutura estática do programa que implementa o modelo ao invés da descrição da estrutura do modelo. Isso facilita a compreensão do mesmo na medida em que permite visualizar os diferentes tipos de relacionamentos que envolvem as entidades (inclusive os de herança) bem como permite a criação de

"templates" de modelos como se pode ver no exemplo do modelo do estabelecimento de lavagem de veículos do capítulo 5 o que pode ser útil para o teste de diferentes configurações. Pela mesma razão, o reuso das classes é facilitado. Um formalismo gráfico para a descrição do comportamento das entidades, assim como o adotado por SMOOCHES, é uma falha que deve ser reparada em uma futura extensão de SIMOO.

TABELA 7.1 - Análise comparativa

	SIMOO	BIB. CLASSES	SHEW CHUCK CHANG	PRISM	DOSE	MODSIM II	MODES	SIM+	VISE	BIB. CLASSES	KOCH LANG	EXSIM	BIB. CLASSES	ABEL JUD	VMSS	SMOOCHES	VSE
Projeto hierárquico	x	-	x	x	x	x	-	-	x			x	x		-	x	x
Múltiplos paradigmas	x	-	-	-	-	-	-	-	-			-	-		-	-	-
Representação gráfica da estrutura estática do modelo	x	-	x	-	-	-	-	-	-			x	-		x	x	x
Representação gráfica do comportamento das entidades	-	-	-	-	-	-	-	-	-			-	-		-	x	-
Ambiente para modelagem	x	-	x	-	-	-	-	-	-			x	-		x	x	x
Recursos para interação entre o usuário e o modelo	x	-	x	-	-	-	-	x	x			-	x		x	-	x
Recursos para visualização de resultados e animação	x	-	x	x	x	-	-	x	x			-	x		x	-	x
Recursos para coleta de estatísticas		x	x	x	x	x	x	x	x			x	x		x	x	x
Possibilidade de execução em ambiente distribuido	x	-	-	-	x	x	x	-	-			-	-		-	x	-
Incentivo a criação de bibliotecas de entidades	x	-	x	-	-	-	-	-	-			x	-		-	x	x

Quase todas as ferramentas descritas incentivam a criação de bibliotecas de entidades. SIMOO e VSE, entretanto, são os únicos que possuem recursos específicos para tanto. VSE permite a criação de bibliotecas e permite documentar as dependências de um modelo em relação a uma biblioteca. SIMOO, através das opções "exportar" e "importar" de seu editor de modelos, oferece uma forma simples e flexível de reuso que permite reusar inclusive partes de hierarquias de herança.

Embora a versão atual de SIMOO execute sobre um sistema operacional que simula o paralelismo, os modelos nele desenvolvidos podem executar sem alterações em plataformas distribuídas. Entre as plataformas estudadas, apenas Sim++ e SIMOO previnem erros de causalidade. Embora a estratégia conservativa adotada em SIMOO para prevenir este erro reduza o grau de paralelismo obtido durante a execução do modelo (Sim++ adota uma estratégia otimista que prevê eventos atrasados), esta estratégia simplifica a construção do mesmo. Assim como Sim++ e os outros ambientes que oferecem a possibilidade de execução em ambientes distribuídos (ModSim II, MODES e SMOOCHES), SIMOO não oferece recursos para a prevenção de "dead-locks".

Quanto aos recursos para a coleta de estatísticas, a totalidade dos ambientes analisados oferece recursos básicos para geração de distribuições de probabilidade, controle das seqüências de números pseudo-aleatórios e coleta de estatísticas. Um recurso adicional oferecido por SIMOO são os monitores que permitem separar a coleta de estatísticas da descrição do comportamento das entidades. Entre os ambientes estudados, porém, apenas a biblioteca de classes de Kocher e Lang e VSE implementam recursos para lidar com múltiplas reproduções independentes do modelo e períodos de "warm-up" assim como cálculos de médias e desvio padrão.

SIMOO também é o único que permite a utilização de vários paradigmas de simulação em um único modelo e ainda incentiva a derivação de novos paradigmas a partir dos já existentes. PRISM é o único, dentre os estudados, que levanta esta última hipótese como possibilidade.

Em relação a recursos para interação entre o usuário e o modelo, SIMOO, PRISM e VISE são os que oferecem os maiores recursos. PRISM oferece mais liberdade no que diz respeito à alteração do comportamento do modelo porque permite alterações em suas planilhas simbólicas durante a execução. VISE é mais limitado porque utiliza Simscript como linguagem de modelagem e não tem como interferir junto às estruturas internas do modelo. O sistema de troca de mensagens de SIMOO impõe poucos limites sobre as possibilidades de interação entre o usuário e o modelo.

A maioria dos ambientes oferece algum tipo de recursos para a visualização do andamento da simulação e de resultados parciais. SIMOO e VISE, entretanto são os únicos que implementam a idéia de associação dinâmica de representações visuais a entidades ou seus atributos [FRE94].

Desta forma pode-se concluir que SIMOO destaca-se em relação aos demais ambientes principalmente por três aspectos:

- pelo conjunto de recursos agrupados de maneira harmônica em um mesmo ambiente;
- pela forma diferenciada de representar os modelos;
- pela abordagem de múltiplos paradigmas.

Além disso, a criação de bibliotecas de componentes controladas por elementos monitores permite transformar SIMOO em um meta-ambiente destinado a criação de ambientes de simulação de propósitos específicos.

Por fim, sua arquitetura simplificada baseada na troca de mensagens entre elementos autônomos facilita a extensão do sistema tanto no que diz respeito a criação de novos paradigmas como recursos para interação com o usuário ou análise estatística.

8 Conclusões e Trabalhos Futuros

8.1 Considerações gerais

Este trabalho discutiu o uso de orientação a objetos em simulação, a ortogonalidade entre os conceitos de orientação a objetos e os paradigmas de simulação, bem como uma forma de se aproveitar essa relação na construção de um sistema de simulação de propósitos gerais. Aspectos complementares tais como simulação distribuída e recursos de visualização e interação com o usuário foram incluídos na discussão.

A maioria dos autores apresenta suas idéias baseadas em um único paradigma de simulação. Não existe grande preocupação em distingüir o paradigma de projeto/implementação, no caso orientação a objetos, do paradigma de simulação. Como conseqüência essas abordagens não oferecem a flexibilidade de permitir a escolha do paradigma mais adequado ao problema a ser solucionado. Além disso, pode-se observar que os ambientes que permitem maior grau de interação entre o usuário e o modelo são os de propósitos específicos. Quanto maior a generalidade do sistema o grau de interação tende a diminuir.

SIMOO é um "framework" para simulação discreta que se distingüe de outras abordagens por permitir a seleção do paradigma mais adequado à descrição de cada entidade particular do modelo. Esta seleção leva em conta os diferentes aspectos que compõem cada paradigma. SIMOO faz um distinção clara entre o paradigma de projeto/implementação, no caso orientação a objetos, e os vários paradigmas tradicionais de simulação que ele suporta. SIMOO compreende uma biblioteca de classes para simulação e um editor de modelos.

A identificação dos aspectos comuns dos paradigmas tradicionais de simulação permitiu a criação do elemento autônomo básico. Este encapsula o controle de uma "thread" própria de execução e um sistema de troca de mensagens não tipadas sincronizadas por um relógio simulado. A grande vantagem de seu uso é que se trata de uma abstração genérica que suporta a derivação de vários paradigmas de simulação. Além disso, o uso de mensagens não tipadas e a autonomia de execução facilitam a construção de um ambiente que admite um alto grau de interação entre o usuário e o modelo. O elemento autônomo é a unidade oferecida pela biblioteca de classes de SIMOO a partir da qual são derivadas todas as entidades do modelo.

A construção de um ambiente que permite um alto grau de interação entre o usuário e o modelo foi simplificada, também, pelo uso de uma meta-classe e de meta-objetos na implementação da biblioteca. A meta-classe funciona como um repositório de dados sobre a estrutura de cada tipo de entidade, além de aramazenar referências para todas as instâncias presentes em um dado instante no modelo. Além disso, oferece uma interface por troca de mensagens que permite a criação e remoção de entidades. Os meta-objetos são capazes de fornecer informações sobre o estado das entidades correspondentes, bem como permitir alterações neste estado ou mesmo mudanças no comportamento das entidades. Desta forma o usuário pode investigar ou alterar qualquer aspecto de qualquer entidade do modelo durante a simulação.

Outro tipo de estrutura derivada a partir do elemento autônomo básico são os monitores. Estes são de grande valia para manter a independência entre os diferentes domínios envolvidos na construção de um modelo de simulação. Através do uso de monitores pode-se manter a descrição do modelo restrita à lógica do mesmo, independente de aspectos tais como visualização ou coleta de estatísticas.

Além de oferecer o elemento autônomo básico, SIMOO incorpora um conjunto de abordagens básicas que permitem a configuração de 4 paradigmas de simulação distintos. Esse conjunto inicial é suficiente para a construção de uma grande variedade de modelos, porém pode ser facilmente estendido se necessário.

O editor de modelos permite a representação gráfica da estrutura de classes do modelo bem como sua particularização através de um diagrama de instâncias. Os tipos de relacionamentos que podem ser definidos entre as classes, em especial a relação de "conhecimento", bem como o uso de um diagrama de instâncias, são adequados a uma ferramenta de implementação de modelos de simulação. Uma das preocupações durante o desenvolvimento de SIMOO foi o de criar um ambiente de implementação que garantisse uma transição suave entre as etapas de projeto e implementação.

A edição do diagrama de classes pode ser feita de forma hierárquica, permitindo o detalhamento do modelo em diferentes níveis de acordo com a necessidade. Além de representar a estrutura de classes do modelo, o diagrama de classes é utilizado para a navegação pelo modelo simplificando a localização de trechos específicos em modelos muito grandes. Para cada nível do diagrama de classes um diagrama de instâncias deve ser especificado. O uso do diagrama de instâncias permite esclarecer situações que podem permanecer dúbias no diagrama de classes. Além disso o diagrama de classes pode ser visto como uma especificação mais genérica a partir da qual diversos modelos podem ser instanciados.

Para cada classe do diagrama de classes formulários adequados podem ser abertos para permitir a descrição do comportamento do tipo de entidade correspondente usando-se um sub-conjunto da linguagem C++. O uso de C++, embora visto com restrições por muitos autores, facilita a portabilidade do sistema. Finalmente, opções específicas facilitam o reuso das classes entre diferentes modelos, apoiando a criação de bibliotecas de entidades.

O conjunto de características de SIMOO o transforma em uma ferramenta de grande flexibilidade que pode ser usada tanto para o desenvolvimento de modelos específicos como para a construção de ambientes configuráveis. Atualmente modelos de processadores com finalidade educacional vêm sendo desenvolvidos dentro do contexto do projeto T&D Bench [FER97] no Curso de Pós Graduação em Ciência da Computação da UFRGS. Um ambiente configurável para a simulação de sistemas de manufatura vem sendo desenvolvido como trabalho de graduação do curso de Bacharelado em Informática da PUCRS. O ambiente vem sendo usado, também, para a modelagem de sistemas de tempo real no contexto do projeto ADOORATA desenvolvido no Departamento de Engenharia Elétrica da UFRGS. Por fim, dois novos trabalhos de graduação que utilizam o ambiente SIMOO estão em fase inicial de projeto no Instituto de Informática da PUCRS. Um pretende desenvolver uma ferramenta

configurável para testes relativos a aeroportos e o outro utilizará SIMOO para o teste de algorítmos inteligentes para o controle de robôs em sistemas de manufatura.

8.2 Continuidade do trabalho

Embora um protótipo de SIMOO encontre-se funcionando e sendo usado em diferentes projetos, muitos aspectos podem ser aprimorados.

SIMOO é fraco em recursos tanto para análise de dados de entrada, determinação de distribuições de probabilidade, correlações, etc como para a organização dos dados resultantes dos diferentes experimentos, cálculo de intervalos de confiança, analise de variância, etc. Recursos nesse sentido devem ser acrescentados e integrados aos recursos de interação já existentes.

Embora a estrutura estática do modelo seja representada graficamente, a descrição do comportamento das entidades ainda é feita textualmente usando-se uma sintaxe similar a C++. Algum tipo de representação gráfica formal do comportamento das entidades poderia facilitar as etapas de verificação e validação dos modelos.

O desenvolvimento dos monitores ainda é uma tarefa complexa. Mesmo a adoção de algum tipo de representação formal para a descrição do comportamento das entidades não irá contribuir para a redução da complexidade da tarefa de se desenvolver os monitores. A verificação da necessidade e viabilidade do uso de diagramas específicos para a modelagem de monitores de visualização, interação ou coleta de estatísticas é uma tarefa que deve ser abordada.

Finalmente, a extensão do conjunto de abordagens de modelagem hoje disponíveis e o transporte de SIMOO para um ambiente realmente distribuído juntamente com recursos que permitam a configuração dessa distribuição são algumas das inúmeras tarefas que aguardam a continuidade do trabalho.

ANEXO - Exemplos

O objetivo deste anexo é detalhar exemplos completos desenvolvidos usando-se SIMOO de maneira a facilitar a compreensão do mesmo. Serão discutidos dois exemplos. O primeiro modela um experimento que visa explicar o conceito de distribuição de probabilidade. O segundo apresenta o modelo de um processador Intel 8051. O primeiro modelo, por sua simplicidade, será detalhado em minúcias. Já o modelo do processador, dado seu tamanho, será apresentado em linhas gerais.

Exemplo 1: Tábua de Testes

1.1 O problema

A tábua de testes é descrita por Weaver [WEA82] e destina-se a facilitar a compreensão do conceito de distribuição de probabilidade. Consiste em uma tábua inclinada onde são lançadas bolinhas (veja figura A1.1). As bolinhas deslizam pela tábua em direção a um conjunto de canaletas onde terminam por se acomodar. No caminho entre o ponto onde são lançadas e as canaletas encontram uma série de obstáculos que as fazem desviar. Cada vez que encontra um obstáculo, uma bolinha tem 50% de chance de desviar sua rota para a direita ou para a esquerda. Repetindo-se o experimento várias vezes e observando-se o posicionamento final de um conjunto de bolinhas nas canaletas tem-se idéia da freqüência com que cada canaleta recebe uma bolinha. Observa-se na verdade, um histograma de freqüência da ocorrência das bolinhas em cada canaleta o que nos dá uma idéia aproximada do gráfico de distribuição de probabilidades que descreve esse comportamento. Segundo Weaver [WEA82], o posicionamento das bolinhas será descrito por uma curva normal.

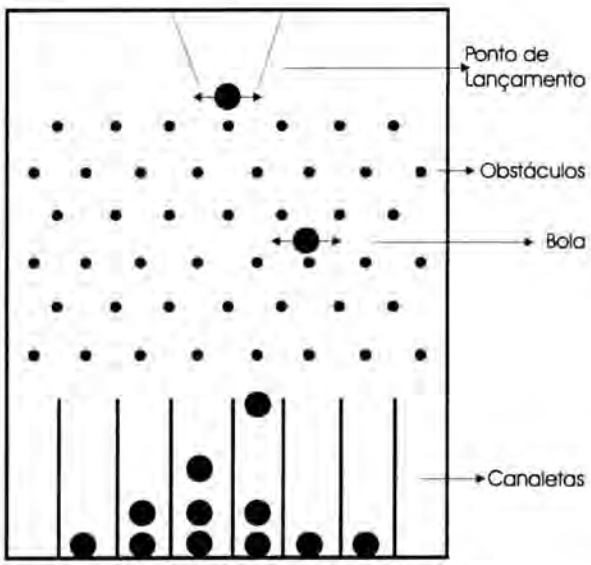


FIGURA A1.1 - Tábua de testes

Além de sua simplicidade, este modelo foi escolhido pela facilidade de validação. Para verificar se o mesmo está correto basta observar as figuras formadas pelas bolinhas após cada experimento e verificar se o desenho formado assemelha-se a uma curva normal.

1.2 Estrutura do modelo

A tábua de testes envolve dois tipos de entidades: a bolinha e a tábua propriamente dita (compreendendo os obstáculos e as canaletas). Para efeitos de modelagem acrescentou-se a essas uma entidade responsável pela geração das bolinhas a intervalos regulares de tempo.

Como todo modelo SIMOO, o nível mais alto da hierarquia do modelo é composto por uma única classe que agrega todo o modelo. A figura A1.2a apresenta o nivel mais alto do modelo. A figura A1.2b apresenta o único nivel de detalhamento.

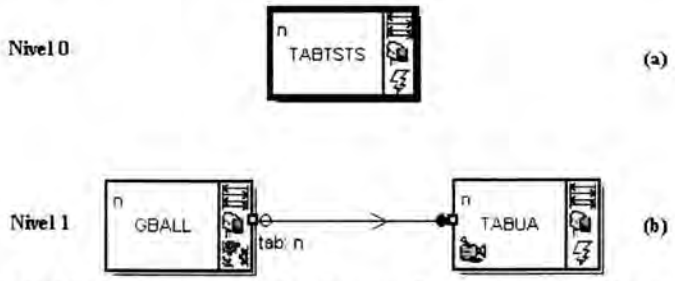


FIGURA A1.2 - Diagrama de classes do modelo da tábua de testes

A classe GBALL é responsável pela geração das bolinhas a intervalos regulares de tempo. A classe tábua mantém a descrição da tábua propriamente dita. Devido a simplicidade deste problema, optou-se por modelar as bolinhas como mensagens. A criação de uma bolinha é sinalizada pelo envio de uma mensagem por uma instância da classe GBALL para uma instância da classe TABUA. A partir de então, a trajetória das bolinhas é controlada pela própria tábua através do envio de mensagens para si própria. Detalhes desse comportamento serão descritos na seção 1.4. O diagrama de instâncias do modelo da tábua de testes pode ser visto na figura A1.3.

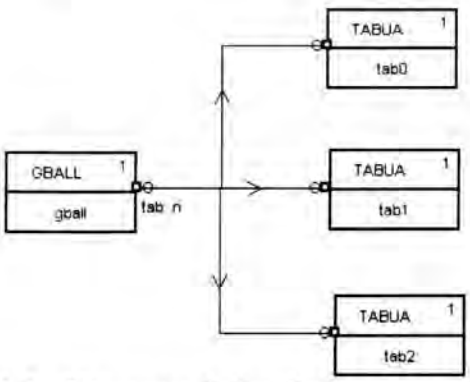


FIGURA A1.3 - Diagrama de instâncias do modelo da tábua

Para a realização de um único experimento, apenas uma instância das classes GBALL e TABUA se fazem necessárias. Neste caso, entretanto, optou-se por instanciar três vezes a classe TABUA como forma de executar três experimentos de cada vez. Este número pode ser ampliado durante a execução pela criação de mais instâncias da classe tábua (observe que o relacionamento entre GBALL e TABUA é 1:n). Uma vez que a geração das bolinhas não leva em consideração fatores aleatórios, apenas uma instância

da classe GBALL se faz necessária (a cada bolinha gerada a mensagem correspondente é replicada para todas as instâncias de tábua conectadas).

1.3 Descrição do comportamento

A classe GBALL é descrita utilizando-se o paradigma de orientação a processos e comunicação por mensagens. A interface dessa classe é composta por uma única mensagem, START, disparada quando da criação da classe e que contém o processo que descreve o ciclo de vida da mesma. A figura A1.4 apresenta o código correspondente à mensagem START da classe GBALL.

```
Parametros par;
int i,j;

// Laço que cria 60 bolas com intervalo de 5 seg. entre elas
for(i=0; i<60; i++)
  {
    Wait(5.0);
    par.SetList("%f%d%d",Clock(),15,1);
    for(j=0; j<tab.Count(); j++)
      {
        m.Set(Id(),tab.Get(j),"NEWBALL",NORMAL_PR,par);
        Send(m);
      }
  }
```

FIGURA A1.4 - Código correspondente à mensagem START da classe GBALL

O processo de criação das bolinhas é muito simples. Foi arbitrada a geração de 60 bolinhas, embora esse pudesse ser um parâmetro de instanciamento.

Um laço do tipo "for" controla a geração das bolinhas. Logo no início do laço, a chamada à rotina "Wait" da biblioteca de SIMOO garante um intervalo de 5 unidades de tempo entre a geração de cada bolinha. Em seguida os parâmetros da mensagem que sinaliza a criação da bolinha são montados. Indicam-se o tempo de criação da bolinha e sua posição inicial sobre a tábua. O tempo de criação será usado como identificador da bolinha, visto que sobre uma mesma tábua não existem duas bolinhas criadas no mesmo instante.

Montados os parâmetros que descrevem a bolinha é necessário enviar a mensagem que sinaliza a criação da mesma para todas as tábuas presentes no modelo. Isso é feito através de um laço tipo "for" interno ao primeiro. A variável "tab" é do tipo REFLIST e representa o relacionamento entre as classes GBALL e TABUA. O tipo REFLIST, definido na biblioteca de classes de SIMOO, é capaz de armazenar uma lista de referências para instâncias e é usado na implementação de relacionamentos do tipo "para muitos" entre classes. O método "Count" de REFLIST retorna a quantidade de referências armazenadas enquanto que o método "Get" retorna a i-ésima referência armazenada. Para cada referência armazenada é montada e enviada a mensagem correspondente. Dessa forma todas as instâncias de TABUA receberão uma nova bolinha. Esta estrutura permite alterações na quantidade de instâncias de TABUA sem que sejam necessárias outras alterações no modelo.

A classe TABUA possui uma interface mais elaborada. É capaz de responder a 3 mensagens: START, NEWBALL e STEP. Seu comportamento é descrito utilizando-se o paradigma de orientação a eventos e comunicação por mensagens.

A mensagem START apenas provê a inicialização dos atributos da classe (figura A1.5).

```
int Tab[32][32];
RN opcao;
int limite;
```

FIGURA A1.5 - Atributos da classe TABUA

O atributo "Tab" corresponde à matriz que descreve a superfície da tábua. Nesta matriz anotam-se as posições ocupadas pelas bolinhas. O atributo "opcao" é uma instância da classe RN da biblioteca de SIMOO. Esta é responsável pela geração de números aleatórios no intervalo [0;1]. A cada chamada do método "NextNumber", um novo valor aleatório é gerado. Finalmente, o atributo "limite" armazena o valor de referência usado para determinar a trajetória da bolinha a cada obstáculo. Inicialmente seu valor é 0.5 indicando 50% de chance de desvio para a esquerda ou direita. Este valor pode ser alterado durante a execução de maneira que se possa observar as implicações na disposição final das bolinhas.

A figura A1.6 apresenta o código correspondente à mensagem START da classe TABUA.

```
int i,j;

// Inicializando a matriz:
for(i=0; i<22; i++)
  for(j=0; j<32; j++)
    tab[i][j] = LIVRE;

// Inicializa a variavel que determina a probabilidade de
// desviar para cada lado
limite = 0.5;
```

FIGURA A1.6 - Código correspondente à mensagem START da classe TABUA

A mensagem NEWBALL trata as chegadas de bolinhas na tábua. O código correspondente pode ser visto na figura A1.7. Seu único objetivo é providenciar o início imediato do deslocamento da bolinha sobre a tábua.

```
Parametros par;

// Programa o avanço imediato da bola
par.SetList("%f%d%d",m.GetDados().GetParAsReal(0),
                     m.GetDados().GetParAsInt(1),
                     m.GetDados().GetParAsInt(2));
m.Set(Id(),Id(),"STEP",NORMAL_PR,par);
Send(m);
```

FIGURA A1.7 - Código correspondente a mensagem NEWBALL da classe TABUA

Finalmente, a mensagem STEP é responsável por controlar cada deslocamento da bolinha sobre a tábua. O código correspondente pode ser visto na figura A1.8.

```
Parametros par;
int l,c,al,ac,inc;

// obtem a coluna e a linha atuais
c = m.GetDados().GetParAsInt(1);
l = m.GetDados().GetParAsInt(2);

// Armazena a posição atual
al = l;
ac = c;

// Se esta na última posição do tabuleiro, então não avança mais
if (l == 31) return;

// Se estiver acima da linha 12 calcula desvio lateral
inc = 0;
if (l < 12)
  {
    // Sorteia incremento lateral
    if (opcao.NextNumber() <= limite) inc = 1;
    else inc = -1;

    // Se vai bater nas bordas, vira para o outro lado
    if ((c+inc < 0) || (c+inc > 31))
      inc *= -1;
  }

// Avança se for o caso
if (tab[l+1][c+inc] == LIVRE)
  {
    c += inc;
    l++;
    tab[l][c] = OCUPADO;
    tab[al][ac] = LIVRE;
  }
else return;

// Programa proximo avanço da bola em 1 segundo
par.SetList("%f%d%d",m.GetDados().GetParAsReal(0),c,l);
m.Set(Id(),Id(),"STEP",NORMAL_PR,par);
Send(1.0,m);
```

FIGURA A1.8 - Código correspondente a mensagem STEP da classe TABUA

Inicialmente obtém-se a posição atual da bolinha. Se esta chegou ao fim da tábua, então o movimento se encerra. Caso contrário um novo avanço deve ser calculado. Se a bolinha encontra-se na zona dos obstáculos, então é necessário determinar um deslocamento lateral. Havendo deslocamento lateral ou não a bolinha avança uma casa à frente. Calculada a nova posição verifica-se se a posição calculada já não se encontra ocupada. Se estiver ocupada o movimento se encerra. Se estiver livre ocupa-se a nova posição, libera-se a anterior e programa-se o próximo movimento para uma unidade de tempo no futuro. O movimento pode ser encerrado no caso de uma bolinha tentar avançar para uma posição ocupada porque isso só irá ocorrer nas canaletas. Dado o intervalo de tempo entre a criação de duas bolinhas consecutivas (5 unidades de tempo) e entre um passo e outro de uma mesma bolinha (uma unidade de tempo) é impossível que elas se encontrem antes de seu destino final.

1.4 Recursos de Visualização

O modelo descrito nas seções 1.2 e 1.3 não prevê a visualização do modelo, o que neste caso é fundamental. Como descrito no capítulo 3, SIMOO oferece o recurso dos monitores permitindo manter a descrição do modelo restrita à lógica do mesmo, independente dos recursos de visualização. No caso do modelo da tábua de testes um único monitor é suficiente. A figura A1.9 apresenta os diagramas de classe e de instâncias com a presença do monitor de visualização.

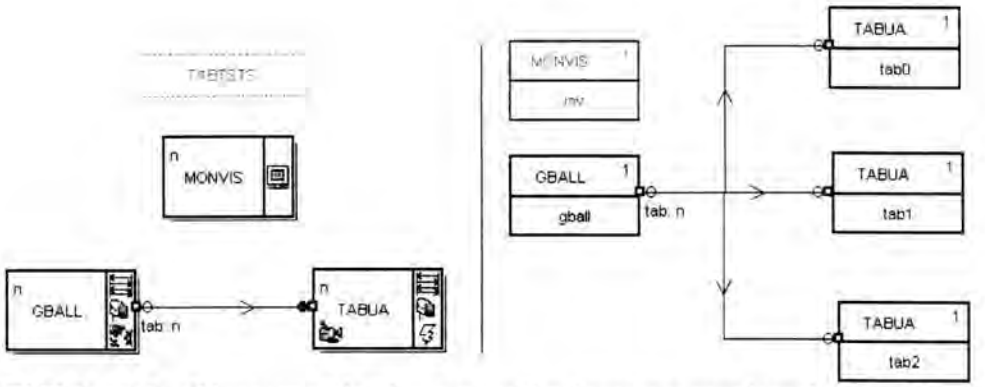


FIGURA A1.9 - Diagramas de classe e instância com o monitor de visualização

Como visto no capítulo 3, um monitor de visualização é um elemento autônomo orientado a eventos com comunicação por mensagens, que recebe uma cópia das mensagens recebidas pelas instâncias que esta monitorando. Observando-se o diagrama de classes percebe-se que apenas as instâncias de TABUA são observadas por este monitor (observe o ícone da câmera no canto inferior esquerdo da representação da classe).

A interface do monitor de visualização é capaz de atender a três mensagens: M_NEWBALL, M_STEP e SIMOO_STARTMONITOR. Toda a mensagem antecedida por "M_" é, na verdade, cópia de uma mensagem recebida por um dos elementos monitorados (pode-se descobrir para quem a mensagem original se destina pela análise do campo "destino" da mesma). Desta forma sempre que o monitor receber as mensagens "M_NEWBALL" e "M_STEP" saberá que são cópias de mensagens "NEWBALL" e "STEP" recebidas por uma das instâncias de tábua. A mensagem SIMOO_STARTMONITOR é enviada por cada um dos elementos monitorados para dar ciência ao monitor de que o elemento a ser monitorado foi criado. Ela pode ser sobrecarregada para que se adicione um comportamento específico. No caso em questão esta mensagem foi sobrecarregada para criar a representação gráfica das tábuas que forem criadas. O código correspondente pode ser visto na figura A1.10.

```
Parametros par;
CString gclass,aux;
int n;

// Pega o número que encerra o nome da tabua
n = PegaNro(m.GetIdOrig());

// Se n estiver fora de faixa, exibe msg de erro e encerra
if ((n<0) || (n>10)) MsgErro("Quantidade de tabuleiros invalida");

// Cria rep grafica do tabuleiro:
// Cria o elemento de visualizacao da cena
aux.Set("#VisTab");
aux.ConCat(n);
vistab[n] = aux;
gclass = "ANIMATE";
par.Set("Tabua");
new ANIMATE(Id(),gclass,&vistab[n],n*200,200,par,168,347);
```

FIGURA A1.10 - Código correspondente à mensagem SIMOO_STARTMONITOR da classe MONVIS

Quando o monitor recebe uma mensagem SIMOO_STARTMONITOR, a criação de uma representação gráfica - usando os recursos da biblioteca de SIMOO - é providenciada. Para compor o identificador desta representação utiliza-se o último caracter do identificador do elemento autônomo que corresponde à tábua sendo monitorada. Este fato implica que as tábuas sejam nomeadas respeitando-se essa regra (observe os identificadores das instâncias de TABUA na figura A1.9), porém simplifica a implementação do monitor. A rotina "PegaNro" (figura A1.11) encarrega-se de extrair o valor correspondente ao último caracter do identificador da instância correspondente.

```
int PegaNro(Cstring n)
  {
    int aux;

    aux = (int)(n.GetC(n.Len()-1)-'0');
    return(aux);
  }
```

FIGURA A1.11 - Código da rotina "PegaNro" da classe MONVIS

Sempre que uma bolinha é criada, uma representação gráfica correspondente deve ser gerada. Isso é providenciado quando o monitor recebe uma cópia da mensagem NEWBALL recebida pela tábua correspondente. O código que providencia esta representação gráfica pode ser visto na figura A1.12.

```
Parametros par;
CString idrg;
int x,y,n;
// Pega o número que encerra o nome da tabua
n = PegaNro(m.GetIdDest());
// Cria rep. grafica da bola:
// Calcula posicao da bola
CalcPos(m.GetDados(),&x,&y);
// Cria representacao grafica da bola
idrg.Set("B");
idrg.ConCat(m.GetDados().GetPar(0));
par.SetList("%S%s%d%d%d",idrg,"Ball",1,x,y);
m.Set(Id(),vistab[n],"INSERTICON",NORMAL_PR,par);
Send(m);
```

FIGURA A1.12 - Código correspondente à mensagem M_NEWBALL da classe MONVIS

A rotina "CalcPos" utilizada no tratamento da mensagem M_NEWBALL extrai a nova posição da bolinha dos parâmetros recebidos e calcula a posição correspondente na representação gráfica. O código correspondente encontra-se na figura A1.13.

```
void CalcPos(Parametros par, int *x, int *y)
  {
    int c,l;

    // Obtem a linha e a coluna da matriz
    c = par.GetParAsInt(1);
    l = par.GetParAsInt(2);

    // Calcula a posição na janela de exibição
    *x = ((c * 10) + 2)/2;
    *y = (l * 10) + 2;
  }
```

FIGURA A1.13 - Código da rotina "CalcPos" da class MONVIS

Finalmente, sempre que o monitor recebe cópia da mensagem STEP enviada a uma das tábuas presentes no modelo, ele providencia a atualização da representação gráfica da bolinha. O código correspondente pode ser visto na figura A1.14

```
Parametros par;
CString idrg;
int x,y,n;

n = PegaNro(m.GetIdDest());
CalcPos(m.GetDados(),&x,&y);
// Movimenta representacao grafica da bola
idrg.Set("B"); idrg.ConCat(m.GetDados().GetPar(0));
par.SetList("%S%d%d",idrg,x,y);
m.Set(Id(),vistab[n],"MOVEICONABS",NORMAL_PR,par);
Send(m);
```

FIGURA A1.14- Código correspondente à mensagem M_NEWBALL da classe MONVIS

1.5 Interação com o modelo

Todo o modelo SIMOO dispõe da mesma interface padrão de controle. Esta interface permite, entre outros recursos, controlar o andamento da simulação, inserir ou remover instâncias, alterar valores de variáveis e visualizar e interagir com o relógio de simulação.

A figura A1.15 apresenta a janela de controle padrão com o menu que permite o controle do andamento da simulação.

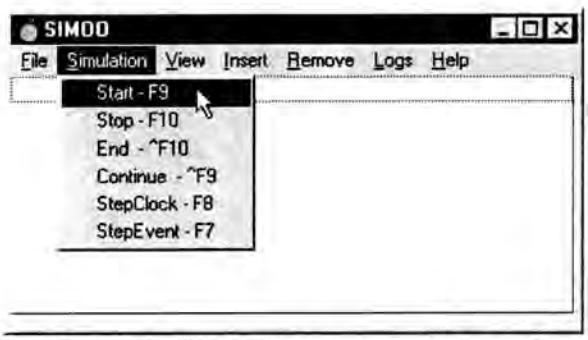


FIGURA A1.15- Interface padrão: menu de controle do andamento da simulação

O menu de controle do andamento da simulação permite iniciar a simulação (opção "Start"), encerrar sua execução (opção "Stop"), preparar para uma nova execução (opção "End"), entrar em modo de execução passo a passo (opções "StepClock" e "StepEvent") e reiniciar a simulação (opção "continue").

A opção "Stop" encerra a execução da simulação mas não remove os elementos autônomos presentes no modelo quando a simulação se encerra. Isso é necessário para permitir que os elementos de interface permaneçam visiveis, em geral exibindo os resultados. A opção "End" elimina todos os elementos autônomos e de interface restantes e prepara o ambiente para que a opção "Start" possa ser executada novamente se for o caso.

As opções "StepClock" e "StepEvent" permitem a execução passo a passo. A opção "StepClock" faz com que a simulação seja interrompida cada vez que o relógio avança. A opção "StepEvent" interrompe a simulação cada vez que um evento é executado em qualquer uma das entidades presentes no modelo. A opção "Continue" retorna ao modo normal de execução.

Como visto na seção A1.4, ao se iniciar a execução do modelo da tábua de testes o monitor de visualização encarrega-se de gerar uma representação gráfica para cada uma das "tábuas" presentes no modelo (figura A1.16).

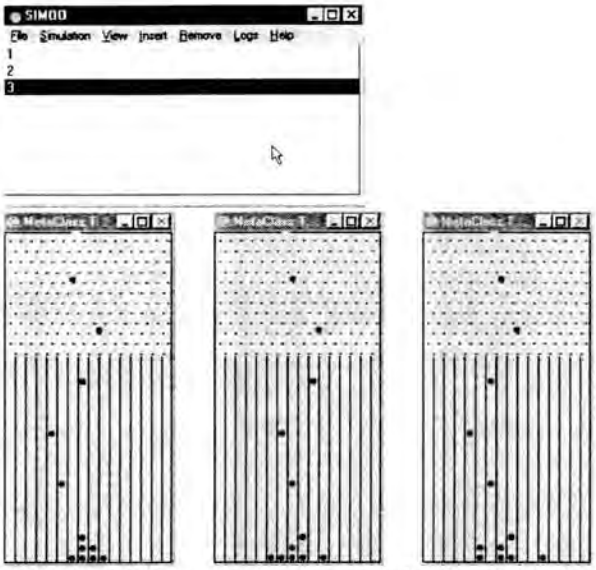


FIGURA A1.16- Execução do modelo da tábua de testes

A figura A1.17 presenta as telas que permitem a visualização das classes e instâncias presentes no modelo bem como a alteração nos valores das variáveis. No caso será alterada a variável "limite" da tábua "tab2", permitindo que se observe a implicação resultante sobre a disposição final das bolinhas.

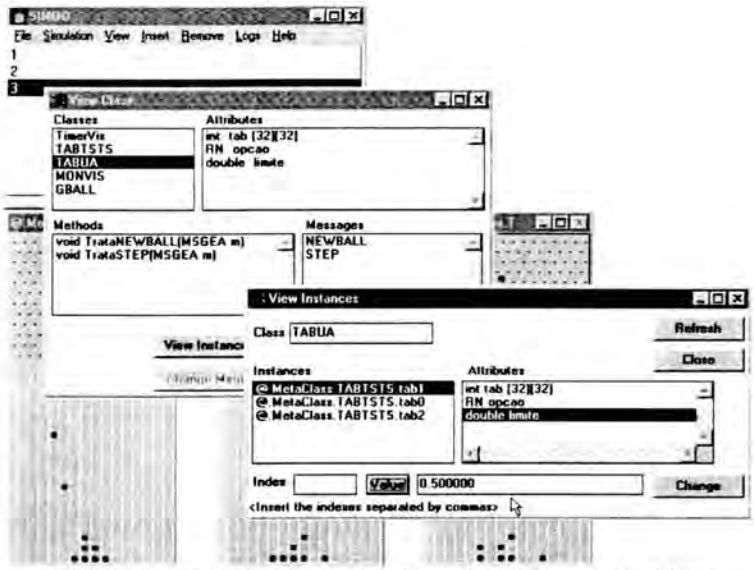


FIGURA A1.17- Alteração do valor de uma variável

A figura A1.18 apresenta as telas que permitem a inserção de novas instâncias e a figura A1.19 a aparência do modelo após a inserção de mais uma "tábua". Observe na figura A1.19 como a disposição das bolinhas se altera após a alteração do valor da variável limite da instância "tab2" de 0.5 para 0.9.

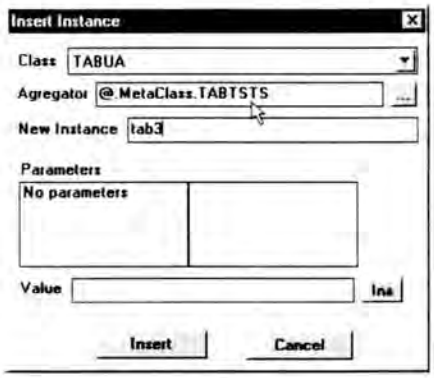


FIGURA A1.18 - Janela de inserção de instâncias

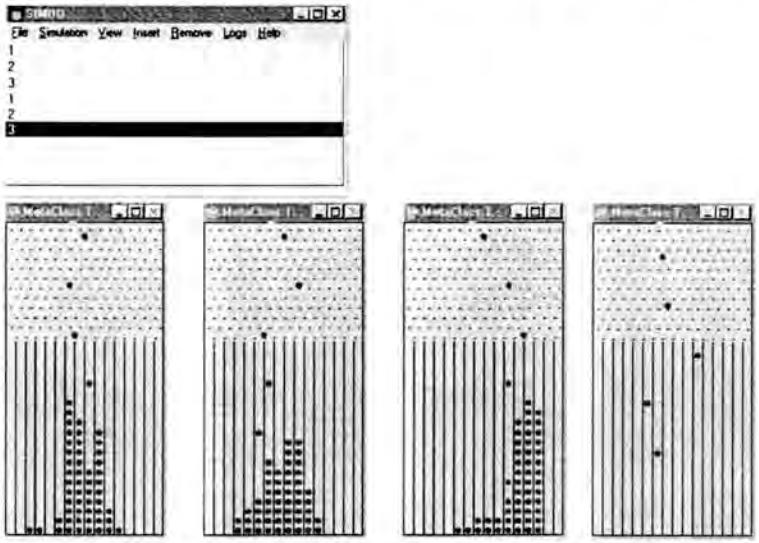


FIGURA A1.19 - Situação do modelo após a inserção de mais uma "tábua"

Finalmente, a figura A1.20 apresenta a janela de interação com a lista de eventos. Através desta, o usuário pode tanto verificar a situação da lista de eventos em um dado instante como acompanhar a sucessão de eventos disparados dinamicamente (veja opção "Automatic refresh"). É possível ainda criar novos eventos, editar os que se encontram na fila ou simplesmente removê-los. Apontando-se um dos eventos da lista com o "mouse", abre-se uma janela que apresenta o mesmo em maiores detalhes.

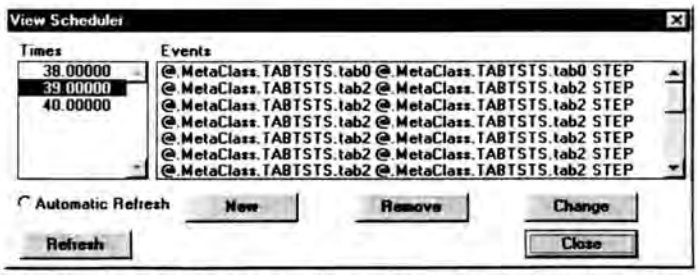


FIGURA A1.20 - Janela de inspeção e alteração do escalonador de eventos

Exemplo 2 - Processador Intel 8051

O modelo do processador Intel 8051 foi construído no contexto do projeto TD-Bench [FER97] com propósitos didáticos. A classe Intel8051 agrega o modelo completo do processador. O primeiro nível de detalhe pode ser visto na figura A1.21. contém 3 classes: Osc, CtrlUnit e BlocFunc. O diagrama de instâncias correspondente contém apenas uma instância de cada classe.

FIGURA A1.21 - Subdiagrama da classe Intel8051

O Intel 8051 foi modelado seguindo-se uma divisão classica para qualquer sistema digital: unidade de controle e bloco funcional. A unidade de controle é modelada pela classe CtrlUnit e a unidade funcional pela classe Blocfunc. A classe Osc modela o oscilador do processador. Pode-se observar que a modelagem desta classe utiliza orientação a processos enquanto que as demais se utilizam de orientação a eventos. Verifica-se aqui a versatilidade de SIMOO em permitir a composição do paradigma mais adequado a descrição de cada entidade. O subdiagrama que detalha a classe BlocFunc pode ser visto na figura A1.22. O diagrama de instâncias correspondente encontra-se na figura A1.23.

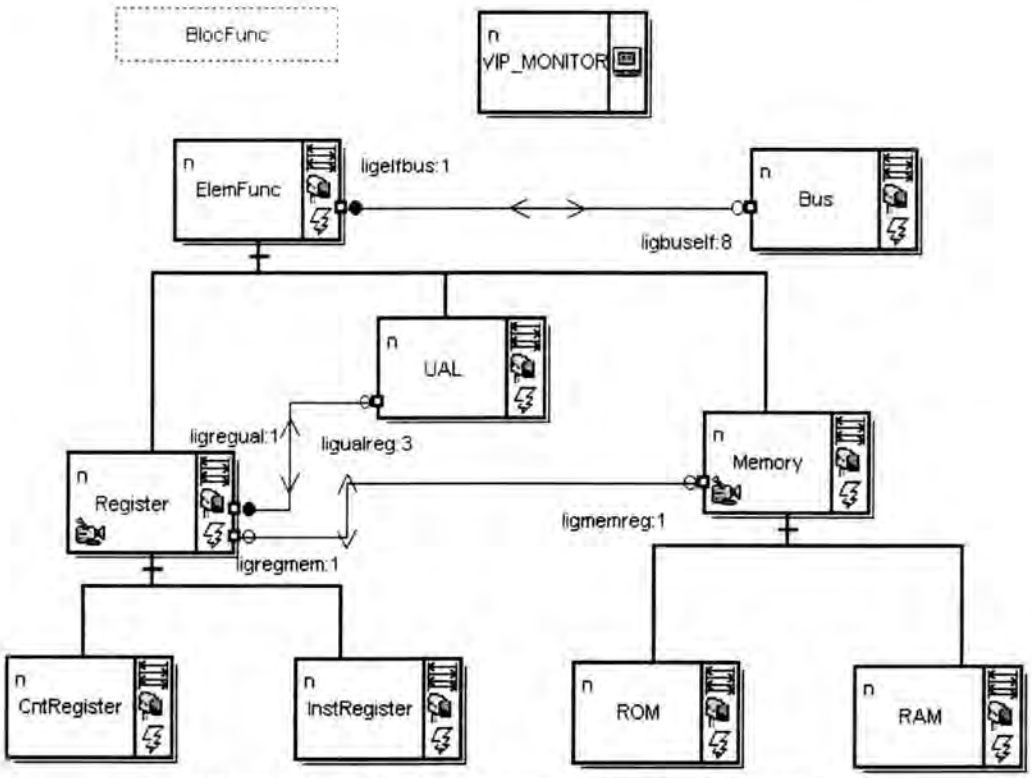


FIGURA A1.22 - Detalhamento da classe BlocFunc

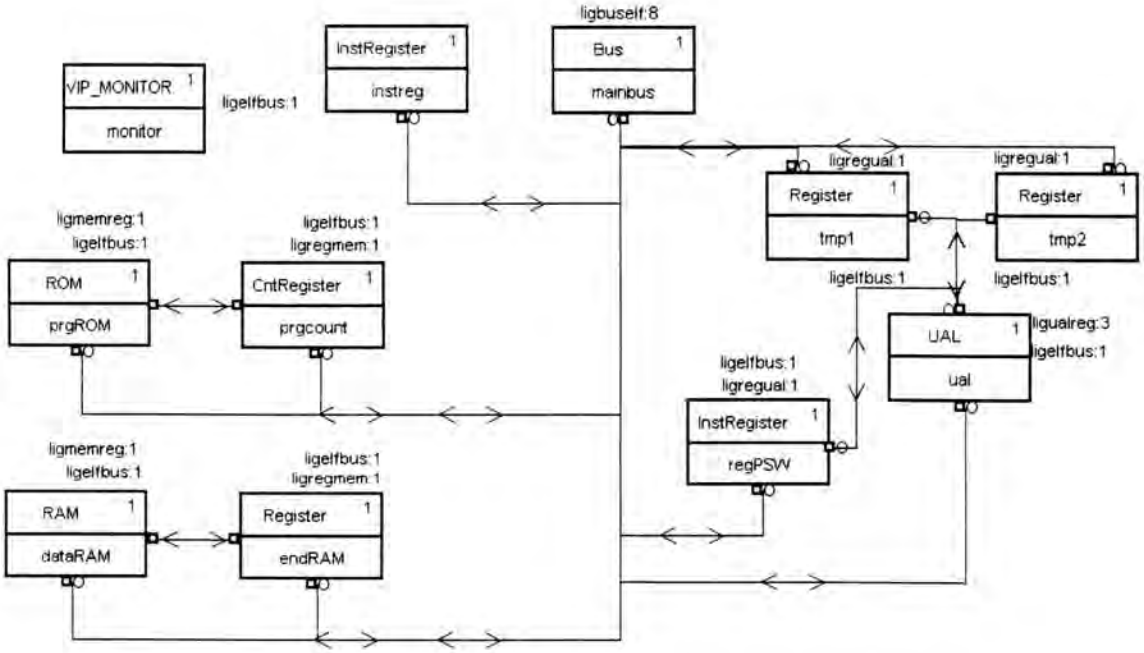


FIGURA A1.23 - Diagrama de instâncias do subdiagrama da classe BlocFunc

A construção das janelas de interação com o usuário, no caso do modelo do processador Intel, foi feita com o auxílio da biblioteca VIP [VIE97]. Esta biblioteca foi especialmente desenvolvida para disponibilizar elementos de interface próprios para a modelagem de processadores. Embora não seja parte integrante da biblioteca básica de SIMOO, pode ser acrescida a qualquer modelo. A figura A1.24 apresenta as principais janelas da interface do modelo do processador Intel.

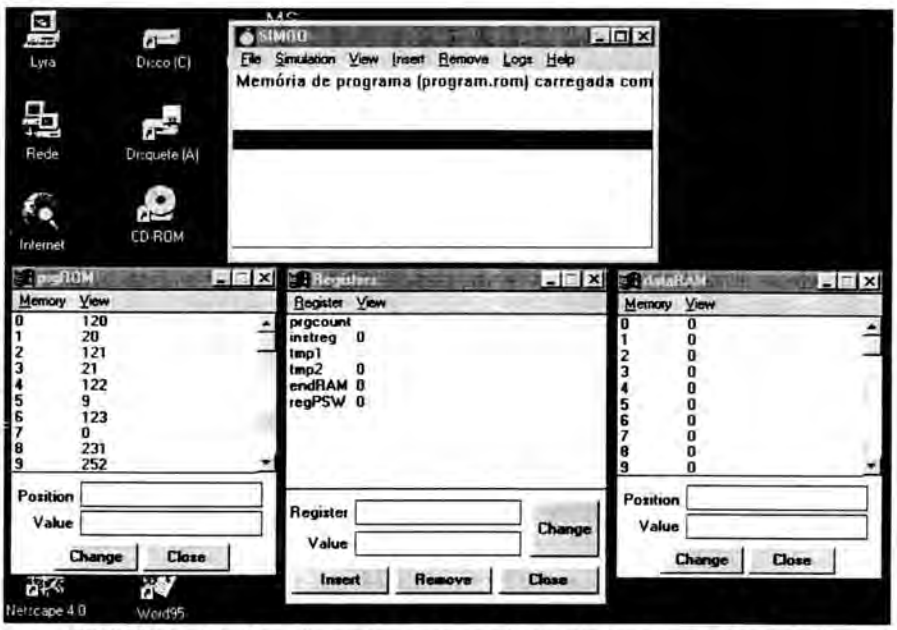


FIGURA A1.24 - Interface do modelo do processador Intel

Bibliografia

[ABE93] ABEL, Jeffrey; JUDD, Robert P. Reusable Simulation Object Specification Through Arbitrary Message Passing and Aggregation Scheme. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1993, La Jolla, CA. **Proceedings**... San Diego: SCS, 1993.

[ARO93] ARONS, Henk de Swaan. Object Orientation in Simulation. In: EUROPEAN SIMULATION SYPOSIUM, 1993, Delft, NL. **Proceedings**... Netherlands: University of Technology, 1993.

[BAE91] BAEZNER, Dirk; LOMOW, Greg. A Tutorial Introduction to Object-Oriented Simulation and Sim++. In: WINTER SIMULATION CONFERENCE, 1991, Phoenix. **Proceedings**... San Diego: SCS, 1991.

[BAL97] BALCI, Osman et al. The Visual Simulation Environment. In: EUROPEAN SIMULATION MULTICONFERENCE, 11., 1997 Istanbul, Turkey. **Proceedings**... San Diego: SCS, 1997. p. 61-68.

[BAR96] BARCIO, B. T. et al. Objetc Oriented Analysis, Modeling and Simulation of a National Air Defense System. **Simulation**, San Diego, v.66, n. 1, Jan 1996.

[BEC93] BECKER, K. **Reusable Frameworks For Decision Support Systems Development.** Namur: Institute D'Informatique - FUNDP, 1993. Ph. D. Tesis.

[BEL90] BELANGER, Ron. MODSIM II: A Modular Object-Oriented Language. In: WINTER SIMULATION CONFERENCE, 1990, New Orleans. **Proceedings**... Lousiana: SCS, 1990.

[BEL87] BELL, P. C.; O'KEEFE, R. M. Visual Interactive Simulation - History, Recent Developments and Major Issues. **Simulation**, San Diego, v. 49 n. 3, p. 109-116, 1987.

[BIG87] BIGGERSTAF, Ted; RICHARD, Charles. Reusability Framework, Assessment and directions. **IEEE Software**, New York, v.4, n. 2, Mar. 1987.

[BIR73] BIRTWISTLE, G. M. **Simula Begin.** New York: Lund Student Litterature, 1973.

[BIS91] BISCHAK, D. P.; ROBERTS, S. D. Object Oriented Simulation. In: WINTER SIMULATION CONFERENCE, 1991, Phoenix. **Proceedings**... Arizona: SCS, 1991.

[BOO94] BOOCH, Grady. **Object Oriented Analysis and Design**. New York: The Benjamin/Cummings Publish Company, 1994.

[BRO90] BROCK, Rebbeca W; WILKERSON, Brian; WIENER, Lauren. **Designing Object Oriented Software**. Englewood Clifs, New Jersey: Prentice Hall, 1990.

[CAC90] Caci Produtcts Company. **Simscript II.5:** Programming Language. La Jolla, CA: [s. n.], 1983.

[CAM96] CAMPO, M; PRICE, R. T. Meta-Objects Manager: A Framework for Customizable Meta-Object Support for Smaltalk-80. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO,1996, Belo Horizonte. **Anais ...** Belo Horizonte: SBC, UFMG, 1996. p. 399 – 413.

[CER94] CERIC, Vlatko Hierarchical Abilities of Diagrammatic Representations of Discret Event Simulation Models. In: WINTER SIMULATION CONFERENCE, 1994. **Proceedings**... [S. l.]: SCS, 1994.

[CHA94] CHANG, W; JONES, L. R. Message Oriented Discret Event Simulation. **Simulation**, San Diego, p. 96 - 104, 1994

[COP96] COPSTEIN, Bernardo; PEREIRA, Carlos Eduardo; WAGNER, Flavio R. The Object Oriented Approach and the Event Simulation Paradigms. In: EUROPEAN SIMULATION MULTICONFERENCE, 10., 1996, Budapest, Hungary. **Proceedings**... [S. l.]: SCS 1996.

[COP97] COPSTEIN, Bernardo; PEREIRA, Carlos Eduardo; WAGNER, Flavio R. SIMOO - An Environment for the Objetc-Oriented Discret Simulation. In: EUROPEAN SIMULATION SYMPOSIUM & EXHIBITION - SIMULATION IN INDUSTRY, 9., 1997, Passau, Germany. **Proceedings**... [S. l.]: SCS, 1997.

[COTA92] COTA, Bruce A.; SARGENT, Robert G. A Modification of the Process Interaction Wold View. **ACM Transactions on Modeling and Computer Simulation**, New York, v. 2, n. 2, p. 109-129, 1992.

[COX87] COX, Springer. Interactive Graphics in GPSS/PC. **Simulation**, San Diego, v. 49, n. 3, p.117-122, 1987.

[EHR79] EHRIG, H. Introduction to the Algebric Theory of Graph Grammars. In: 1st Graph Grammar Workshop, 1., 1979. **Proceedings ...** Berlin: Springer Verlag,1979. P. 1-69. (Lecture Notes in Computer Science, v.73).

[EHK96] EHRIG, H. et al. **Algebric Approaches to Graph Transformations.** [S.l.: s.n.], 1996, v. 1. (The Handbook of Graph Grammars)

[FER97] FERREIRA, Luciano **T&D Bench:** Um Ambiente de simulação Interativa Visual Orientado a Objetos para o Ensino e Projeto de Processadores. Porto Alegre: CPGCC da UFRGS, 1997. Dissertação de Mestrado.

[FIS92] FISHWICK, Paul A. **SimPack: Getting Started With Simulation Programing in C and C++.** Disponivel por www em http://www.cis.ufl.edu/~fishwick (Jul.1996).

[FOO92] FOOT, B. Architetural Blakanization in The Post Linguistic Era. In: WORKSHOP ON OO REFLECTION AND META-LEVEL ARCHITECTURES, OOPSLA, 1993. **Proceedings...** [S.l. : s.n.], 1993. p.1-9.

[FRE90] FREITAS, C. M. D. S. **Técnicas de Visualização em Simulação.** Porto Alegre: CPGCC-UFRGS, 1990. (TI -187).

[FRE94] FREITAS, C. M. D. S. **Uma Abordagem Unificada para Análise Exploratória e Simulação Interativa Visual.** Porto Alegre: CPGCC da UFRGS, 1994. Tese de doutorado.

[FUJ90] FUJIMOTO, Richard. Paralel Discret Event Simulation. **Communications fo ACM**, New York, v. 33, n. 1, Oct. 1990

[GAM95] GAMMA, Erich et al. **Design Patterns - Elements of Reusable Object Oriented Software".** USA: Addison Wesley, 1995.

[GOG95] GOGG, Thomas et al. **Improve Quality & Productivity With Simulation**. [S.l.]: JMI consulting Group, 1995.

[GRA93] GRANT, T. Beyond Objects: An Agent Based Simulation Tool. In: EUROPEAN SIMULATION SIMPOSIUM, 1993, Delft, NL. **Proceedings...** Netherlands: University of Technology, 1993.

[HIL92] HILL, M. Gourgand D. C. R. Object Oriented Modelling and Design Methodology for Symulation Animation. In: EUROPEAN SIMULATION MULTICONFERENCE, 1992. **Proceedings ...** York, United Kingdon: SCS, 1992.

[JAV93] JÁVOR, A. B.; MORÉ, G. Object Oriented Mapping of Real World Systems Into Simulation Models. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1993, La Jolla, CA. **Proceedings...** San Diego: SCS, 1993.

[JEF96] JEFF Duntemann, Jeff, MISCHEL, Jim; TAYLOR, Don. **Delphi 2 Programming.** New York: Coriolis Group Books, 1996.

[JOH91] JOHNSON, R. E.; RUSSO, Vincent F. **Reusing Object-Oriented Design.** Disponível por e-mail em russo@cs.purdue.edu (AGOSTO, 1991).

[JOI94] JOINES, Jeffrey A; ROBERTS, Stephen. Design of object oriented simulations in C++. In: WINTER SIMULATION CONFERENCE, 1994. **Proceedings...** [S. l.: s. n.], 1994

[KAM96] KAMIGAKI, Tamotsu, NAKAMURA, Nobuto. Na Object Oriented Visual Model-Builduing and Simulation System for FMS Control. **Simulation**, San Diego, v. 67, n. 6, p. 375-385, 1996.

[KIM92] KIM, Tag Gon; PARK, Sung Bong. The DEVS Formalism: Hierarquical Modular Systems Specification in C++. In: EUROPEAN SIMULATION MULTICONFERENCE. **Proceedings ...** York, United Kingdon: [s.n.] 1992.

[KRA88] KRASNER, Glenn, E.; POPE Stephen T. A Cookbook for Using The Model-View-Controler User Interface Paradigm in Smalltalk-80. **JOOP**, [S.l.], Aug./Sept. 1988.

[KOC94] KOCHER, Harmut, LANG, Martin An Object-Oriented Library for Simulation of Complex Hierarchical Systems. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1994. **Proceedings...** [S. l.]: SCS, 1994.

[KOR96] KORFF, M.; RIBEIRO, L. Formal Relationship Between Graph Grammars and Petri Nets. In: WORKSHOP ON GRAPH GRAMMARS AND THEIRS APLLICATION TO COMPUTER SCIENCE, 1996. **Proceedings...** Berlin, Springer Verlag, 1996. p. 288-303.

[LAW91] LAW, Averill; KELTON, David, W. **Simulation Modeling and Analysis.** New York: McGraw Hill, 1991.

[LEE93] LEE, Edward A.; KALAVADE Asawaree A Hardware Software codesign Methodology for DSP Applications. **IEEE Design & Test of Computers**, New York, Sept. 1993.

[LIN95] LINDSTAEDT, Ernesto; WAGNER, Flávio Rech. Um Ambiente de Simulação Visual Interativa. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, SEMISH, 22., 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995.

[LIS95] LISBOA, Maria Lúcia Blanck. **MOTF - Meta-Objetos para Tolerância a Falhas.** Porto Alegre: CPGCC da UFGRS, 1995. Tese de doutorado.

[LIS96] LISBOA, Maria Lúcia Blanck. **Reflexão Computacional no Modelo de Orientação a Objetos.** Porto Alegre: CPGCC da UFRGS, 1996. Relatório de Pesquisa.

[LÖW93] LÖWE, M. Algebric Approach to Single-Pushout Graph Transformation. **Theoretical Computer Science,** Amsterdam, n. 109, p. 181-224, 1993,.

[MAK91] MAK, Victor W. DOSE: A Modular and Reusable Object Oriented Simulation Enviroment. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1993, La Jolla, CA. **Proceedings...** San Diego: SCS, 1993.

[MAR90] MARSHALL, R. et al. Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie. **Computer Graphics**, Oxford, v. 24, n. 2, mar. 1990.

[MAT94] MATSUMOTO; KURITA TT800 a Twisted GFSR Generator. **ACM Transactions on Modelling and Computer Simulation**, New York, v. 4, n. 3, p. 254-266, 1994.

[MEY88] MEYER, Bertrand. **Object Oriented Software Construction.** New York: Prentice Hall, 1998.

[MOC92] MOCHEL, Thomas; OBERWEIS, Adreas. An Object Oriented Concept for the simulation of Embedded Systems. In: EUROPEAN SIMULATION MULTICONFERENCE, 1992. **Proceedings ...** York, UK: SCS, 1992.

[NIE93] NIERSTRASZ, Oscar. **Visual Scripting Towards Interactive Construction of Object Oriented Applications.** [S.l. : s.n.], 1993.

[PER 96] PEREIRA, C. et al Object-Oriented Development of Real-Time Industrial Automation Systems. In: IFAC TRIENNIAL WORLD CONGRESS, 1996, San Francisco, USA. **Proceedings...** [S. l.: s. n.], 1996. p. 321-326.

[PER94] PEREIRA, C. E. Real-Time Active Objects in C++/Real-Time UNIX. In: ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILER, AND TOOL SUPPORT FOR REAL-TIME SYSTEMS, 1994, FL, Orlando. **Proceedings...** [S. l : s.n.], 1994.

[PID94] PIDD, Michael. An Introduction to Computer Simulation. In: WINTER SIMULATION CONFERENCE, 1994. **Proceedings...** [S. l.: s. n], 1994.

[RIB96] RIBEIRO, L. **Parallel Composition and Unfolding Semantics og Graph Grammars.** Berlin: Technische universität Berlin, 1996. Phd Thesis.

[ROO91] ROOKS, M. A Unified Framework for Visual Interactive Simulation. In: WINTER SIMULATION CONFERENCE, 1991, Phoenix. **Proceedings...** San Diego: SCS, 1991.

[RUM91] RUMBAUGH et al. **Object Oriented Modeling and Design.** Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[SEL94] SELIC, Bran; GULLEKSON, Garth; WARD, Paul T. **Real Time Object Oriented Modelling.** [S.l.]: Willey Professional Computing - John Wiley and Sons, 1994.

[SHA75] SHANNON, R. E. **Systems Simulation, the Art and Science.** Englewood Cliffs, N. J.: Prentice Hall, 1975.

[SHE91] SHEWCHUCH, John P.; CHANG, Tien-Chien. An Approach to Object Oriented Discret Event Simulation of Manufacturing Siystems. In: WINTER SIMULATION CONFERENCE, 1991, Phoenix. **Proceedings...** San Diego: SCS, 1991.

[SHL92] SHLAER, Sally; MELLOR, Stephen J. **Object Life Cycles - Modeling the World in States**, Englewood Clifs, New Jersey: Yourdon Press, 1992.

[SOM92] SOMMERVILLE, Ian. **Software Engineering**. New York: Addison Wesley Publishing Company, 1992.

[STR90] STROUSTROUP, Bjarne; ELLIS, Margaret A. **Anottated C++:** Reference Manual. [S.l.] Addison Wesley, 1990.

[TAN94] TANIR, Orial; SEVINC, Suleyman. Definig Requirements for a Standart Simulation Enviroment. **Computer,** New York, 1994.

[VAU91] VAUGHAN, P. W. PRISM: An Object Oriented System Modeling Enviroment in C++. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1991. **Proceedings...** [S. l.]: SCS, 1991.

[VIE97] VIERO, Daniel M.; WAGNER Flavio R. **VIP - Biblioteca de Visualização e Interação para Processadores no Ambiente SIMOO.** Porto Alegre: CPGCC da UFRGS, 1997.

[WAS90] WASSERMAN, A. I.; PIRCHER, P. A.; MULLER, Robert, J. The Object Oriented Structed Design Notation for Software Design Representation. **Computer,** New York, p. 50-63, 1990

[WEA82] WEAVER, Warren. **Lady Luck - The Theory of Probability.** [S.l. :s.n.], 1992. p.260 - 261.

[WIL94] WILHELM, Bob. Structural Implications of Concurrent Execution Models for OO Real Time Systems. **ACM SIGPLAN Notices**, New York, v. 29, n. 10, Oct. 1994. Trabalho apresentado na Conference on Object'Oriented Programming Systems Languages and Applications, 9., 1994, Portland, US.

[ZEI88] ZEIGLER, B.; Concepcion, A. DEVS Formalism: A Framework for Hierarquical Model Development. **IEEE Transactions on Software Enginnering**, New York, v. 14, n. 2, Feb. 1988.

[ZHE93] ZHENG Quing; CHOW, Paul Exsim: A General Purpouse Object-Oriented Environment for Discrete-Event Simulations. In: OBJECT ORIENTED SIMULATION CONFERENCE, 1993, La Jolla, CA. **Proceedings...** San Diego: SCS, 1993.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Simoo: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma

por

Bernardo Copstein

Tese apresentada aos Senhores:

Profa. Dra. Carla Maria Dal Sasso Freitas

Prof. Dr. Carlos Eduardo Pereira (IEE/UFRGS)

Prof. Dr. Luis Henrique Rodrigues (PPGEP/UFRGS)

Profa. Dra. Cecília Mary Fischer Rubira (DCC/UNICAMP)

Vista e permitida a impressão.
Porto Alegre, 16/1/98.

Prof. Dr. Flávio Rech Wagner,
Orientador.