UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Concepção de uma Biblioteca de Classes para Sistemas de Manufatura

por

CARLOS ALBERTO BERTOTTO

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

Prof. Dr. Carlos Eduardo Pereira
Orientador

Prof. Dr. Flávio Rech Wagner
Co-orientador

Porto Alegre, janeiro de 2003

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Bertotto, Carlos Alberto

Concepção de uma Biblioteca de Classes para Sistemas de Manufatura / por Carlos Alberto Bertotto. – Porto Alegre: PPGC da UFRGS, 2003.

102 p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientador: Pereira, Carlos Eduardo; Co-Orientador: Wagner, Flávio Rech.

1. Sistemas de manufatura. 2. Simulação. 3. Orientação a objetos. 4. Automod. I. Pereira, Carlos Eduardo. II. Wagner, Flávio Rech. III. Título.

Agradecimentos

Muitas pessoas dedicaram seu tempo e esforço para a conclusão deste trabalho. Manifesto meus sinceros agradecimentos a todas elas.

Dentre estas pessoas, gostaria de agradecer especialmente ao meu orientador Prof. Dr. Carlos Eduardo Pereira pela adequada condução deste trabalho e pela amizade adquirida nestes anos. Agradeço especialmente também ao Prof. Dr. José Luís Almada Güntzel que teve papel fundamental no meu ingresso ao PPGC.

Meu agradecimento de coração vai a minha querida esposa Rafaella, que sempre esteve ao meu lado em todas as horas, me auxiliando, compreendendo e acreditando.

Agradeço também ao co-orientador Prof. Dr. Flávio Rech Wagner, meus pais, José Carlos e Maria Liane, minhas irmãs Adriane, Carine e Luciane, e a empresa Pigozzi S/A na pessoa de Ivandro Amélio Mariani. Agradeço também ao amigo Leandro Buss Becker e ao grupo de professores do curso de Sistemas de Informação e Ciência da Computação da UNIFRA.

"É muito melhor lançar-se à luta, encontrar triunfo e glória, mesmo expondo-se ao insucesso, do que cerrar fileira com os pobres de espírito, que não lutam muito, nem sofrem muito, porque vivem nesta penumbra cinzenta que não conhece vitória nem derrota."

Franklin Delano Roosevelt

Sumário

Lista de Abreviaturas

AGV	Automated Guided Vehicle
AMS	Automated Manufacturing Systems
AO/C++	Active Objects C++
AS/RS	Automatic Storage/Retrieval System
CAD	Computer-Aided Design
CASE	Computer-Aided Software Engineering
CIM	Computer Integrated Manufacturing
CORBA	Common Object Request Broker Architecture
CRP	Capacity Requirements Planning
DIAMOND	Distributed Architecture for Monitoring and Diagnostics
DSS	Distributed Simulation System
FMS	Flexible Manufacturing System
HoMuCS	Holonic Multi-cell Control System
MfgDTF	Manufacturing Design Task Force
MTBF	Mean Time Between Failure
MRP	Manufacturing Resource Planning
OMG	Object Management Group
OO	Orientação a Objetos
OOP	Object-Oriented Programming
ProPlanT	Production Planning Technology
RT-Java	Real Time-Java
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language

Lista de Figuras

Lista de Tabelas

Resumo

Neste trabalho é proposta uma biblioteca de classes para sistemas de manufatura que visa facilitar a construção de modelos de simulação, permitindo o reuso e agilizando a modelagem. A principal característica deste trabalho é sua abordagem que difere da maioria dos trabalhos correlatos na área. Ela baseia-se na utilização de conceitos de produção bastante conhecidos atualmente, como os roteiros e atividades de produção. Isto permite a criação de novas simulações muito mais rapidamente do que em outras metodologias, já que não são necessárias traduções complexas entre a realidade e as aplicações simuladas. A biblioteca desenvolvida foi validada com a aplicação dos conceitos à modelagem de uma linha de produção de pontas de eixo para tratores, produzidos pela empresa Pigozzi S/A. Além disso, o trabalho discute a possibilidades de integração entre a biblioteca de classes proposta e a ferramenta de simulação de sistemas de manufatura Automod. A simulação do estudo de caso da linha de produção de pontas de eixo, modelada tanto no Automod quanto na biblioteca de classes proposta, permitiu uma comparação quantitativa, o que viabilizou a validação este trabalho.

Palavras-Chave: Sistemas de Manufatura, Simulação, Orientação a Objetos, Automod.

TITLE: “A CLASS LIBRARY CONCEPTION FOR MANUFACTURING SYSTEMS”

Abstract

This work presents a class library for manufacturing systems that aims at facilitating the construction of simulation models, allowing reuse and speeding up the modeling process. This library implements a modeling approach that differs from the majority of similar works in this area. It is based on the application of well know manufacturing concepts, like production routes and activities. It allows the creation of new simulations faster than other methodologies, since complex translations from the reality to simulated applications are not necessary. The development of this library was validated by modeling the production line of tractor parts manufactured by Pigozzi S/A company. Moreover, this work discusses the possible integration between the proposed library and Automod, a simulation tool for manufacturing systems. The production line case study, modeled using both Automod and the proposed class library, allowed a quantitative comparison for the validation of this work.

Keywords: Manufacturing Systems, Simulation, Object-Oriented Paradigm, Automod.

1 Introdução

Desde a Revolução Industrial, quando houve a adição de um motor a vapor em um tear, o homem tem conseguido transferir para as máquinas trabalhos insalubres e repetitivos. O tempo passou e novas atribuições eram dadas às máquinas. Com a criação da linha de montagem por Henry Ford, no início do século XX, as fábricas de manufatura de bens de consumo têm sistematicamente buscado a automatização de suas linhas de produção, com o objetivo de reduzir custos e incrementar as vendas. As mudanças foram, uma a uma, tornando-se cotidiano nestas empresas. Esteiras, máquinas de controle numérico, veículos automatizados e robôs começaram a fazer parte da cadeia manufatureira destas indústrias, aumentando a velocidade de produção e retirando definitivamente o homem de trabalhos potencialmente perigosos. Junto com dispositivos físicos eram introduzidos novos estudos sobre as teorias da produção na tentativa de organizar as tarefas de manufatura que começavam a ficar cada vez mais complexas.

Obviamente, era necessário um controle acurado dos resultados da produção e de todos os sistemas automáticos da planta física no chão de fábrica. Foram criados então os sistemas supervisórios, planejamentos MRP e CRP, cartões Kanban, Just-in-Time, além da integração de sistemas de estoque, vendas e suprimentos em uma cadeia de produção que neste estágio começaram a ficar cada vez mais difíceis de controlar e modificar.

O mercado, por sua vez, começa a se tornar cada vez mais especializado, requisitando lotes de produtos cada vez menores e mais específicos. Houve então a necessidade de sistemas de manufatura altamente configuráveis que permitiam, sem muitas mudanças, produzir ora produto X, ora produto Y, mantendo preços e prazos compatíveis com os conseguidos no passado.

Devido a toda esta revolução na produção, as estratégias de controle em sistemas de manufatura tradicionais disponíveis até então estão tendo dificuldades de acompanhar as mudanças no mercado. Isto implica em uma urgente necessidade de se desenvolver novas estratégias de controle que se ajustem às tecnologias emergentes.

A complexidade envolvida no projeto de sistemas de automação industrial torna necessária a adoção de metodologias de projeto que englobem conceitos como a hierarquia, abstração, modularidade, e que permitam o desenvolvimento de sistemas que atendam aos requisitos de custo, tempo de desenvolvimento qualidade e desempenho exigidos pelo cliente.

Por outro lado, as empresas, principalmente as nacionais, não dispõem de tempo nem recursos para testar novas tecnologias que não estejam completamente assimiladas pelo mercado. Ou seja, muitas vezes as novas tecnologias que ingressam nas fábricas são adquiridas sem o planejamento necessário e são colocadas em produção de modo precipitado, acarretando em problemas sérios como a sub-utilização do recurso.

Apesar da simulação existir muito antes do aparecimento do computador, foi depois da década de 1950 que a mesma ganhou força. A simulação computacional é

atualmente uma importante ferramenta de análise e projeto em todas as áreas do conhecimento humano. O ramo dos sistemas de manufatura é uma das principais aplicações da simulação. Através da simulação em sistemas de manufatura é possível avaliar um projeto de, por exemplo, uma nova fábrica sem que tenha uma só máquina adquirida ou avaliar qual o impacto de um novo torno na linha de produção.

Mais recentemente, vários pesquisadores têm estudado novas formas de simulação usando paradigmas diferentes daqueles aplicados até o momento. Um destes paradigmas é a orientação a objetos. A força do paradigma de orientação a objetos aplicado a simulação tem recebido recentemente considerável atenção por pesquisadores. A correspondência um para um entre objetos no problema real e suas abstrações no software provido por este paradigma, oferecem potencial para uma melhor modelagem. Neste caso, a orientação a objetos está sendo explorada para permitir a construção de softwares de simulação de alta fidelidade para suportar a modelagem e controle de sistemas de manufatura complexos e integrados. Conforme Govindaraj (1990), a simulação é uma das únicas metodologias viáveis para uma avaliação detalhada do comportamento, em tempo de projeto, de sistemas de manufatura complexos e integrados.

1.1 Motivação

Apesar de todos os benefícios da simulação na área de manufatura, geralmente as avaliações custam muito caro e levam um tempo considerável para produzirem resultados. Isto porque, para ser gerado um modelo que possa espelhar a realidade, é preciso uma análise do problema muito bem detalhada e a alocação de projetistas de simulação que sejam especialistas em algum dos ambientes de simulação comerciais disponíveis. Sem isso, qualquer projeto de simulação tem grande chance de fracassar.

Existem vários ambientes de simulação comerciais específicos para os sistemas de manufatura. Estes ambientes apresentam muitas características importantes e permitem construir simulações extremamente poderosas que podem fornecer soluções avançadas em novos projetos através da experimentação de várias configurações alternativas.

Uma destas ferramentas de simulação mais conhecida é o Automod (AUTO SIMULATIONS, 2002). O Automod é uma ferramenta de modelagem e simulação comercial criada pela AutoSimulations. Esta ferramenta permite a criação de simulações voltadas especialmente para a indústria de manufatura e transformação, a partir de estruturas (entidades) pré-definidas que imitam o seu comportamento real. Depois de construído o modelo, o mesmo é compilado e executado em um *runtime* que permite a animação da simulação com gráficos de três dimensões, podendo-se analisar a simulação a partir do desenho e dos vários dados estatísticos fornecidos pelo programa.

O grande problema destas ferramentas é a dificuldade na criação de novas simulações. A modelagem nestes ambientes requer um estudo prévio da ferramenta, necessitando, geralmente, de pessoal altamente especializado, já que os mesmos possuem linguagens proprietárias e conceitos de simulação conhecidos por poucos. Além disso, elas não fazem o uso de conceitos que permitam o reuso e extensões, fazendo com que cada nova simulação deva partir do zero. Analisando o atual panorama da complexidade e da necessidade de novos paradigmas de construção dos sistemas de manufatura, este fato é inaceitável. Deste modo, é necessária a possibilidade da integração das vantagens dos simuladores comerciais com os novos paradigmas que

estão surgindo. Isto acarreta um aumento da qualidade dos modelos produzidos, já que os projetistas preocupam-se com a solução do problema de manufatura sem ter que se ocupar com particularidades dos ambientes de simulação.

1.2 Objetivos

O principal objetivo desta dissertação é o de conceber e implementar uma biblioteca de classes para sistemas de manufatura que permita o desenvolvimento de modelos de simulação para a avaliação de tais sistemas. Componentes básicos destes sistemas devem estar disponíveis, além de poderem ser estendidos através das características de agregação e abstração da orientação a objetos. Um segundo objetivo é o de propor e avaliar a possibilidade de integração desta biblioteca de classes com ferramentas comerciais de simulação, tais como o Automod. O objetivo neste caso é conjugar as facilidades que as ferramentas comerciais oferecem com relação ao suporte na interface com usuário com conceitos de modelagem que facilitem a construção de sistemas complexos, incentivando o reuso e facilitando a extensão.

Esta dissertação propõe-se a atingir os seguintes objetivos específicos:

- Realizar um estudo aprofundado sobre os principais focos de pesquisa na área de simulação em sistemas de manufatura;
- Criar e implementar uma biblioteca de classes para sistemas de manufatura;
- Estudar o simulador de sistemas de manufatura Automod visando a construção de simulações complexas;
- Implementar na linguagem de programação C++ as classes da biblioteca de classes com a pretensão de validá-la;
- Através de um estudo de caso, comparar o uso do simulador Automod com a implementação em C++ do mesmo caso;
- Propor um método de integração da biblioteca de classes com o Automod.

1.3 Organização do Texto

O texto desta dissertação é apresentado em seis capítulos e está dividido da seguinte forma: o capítulo 2 apresenta uma análise do estado da arte discorrendo sobre os mais importantes esforços de pesquisa na área dos sistemas de manufatura. O tema apresentado no terceiro capítulo apresenta a biblioteca de classes que foi o principal objeto de pesquisa durante o desenvolvimento deste trabalho. O capítulo 4 trata da apresentação de um estudo de caso que foi usado na validação da biblioteca de classes apresentada no capítulo 3. O quinto capítulo aborda uma proposta preliminar de integração entre um ambiente comercial de simulação de sistemas de manufatura (no caso o Automod) com a biblioteca de classes proposta nesta dissertação. E, por fim, o capítulo 6 traça as considerações finais, apresentando as conclusões obtidas e ponderando sobre possíveis trabalhos futuros.

2 Análise do Estado da Arte

Em todas as pesquisas envolvendo os sistemas de manufatura, o objetivo é sempre melhorar o nível de automação tornando a produção mais flexível e autônoma, visando a redução nos custos, otimização do processo produtivo e aumento da produção. Vários pesquisadores estão engajados em esforços neste sentido. Alguns dos esforços mais importantes na pesquisa em sistemas de manufatura são apresentados nas seções que seguem.

2.1 Sistemas Multiagentes Aplicados aos Sistemas de Manufatura

Uma das características desejadas dos sistemas de manufatura é a descentralização do controle das diversas entidades que compõe este sistema. Um meio de se obter esta descentralização é utilizando os Sistemas Multiagentes. Um agente é uma entidade computacional autônoma que pode perceber seu ambiente por meio de sensores e agir sobre este ambiente por meio de atuadores. Conforme estudos apresentados em Bertotto (2001), esta é uma área promissora de pesquisa e aplicação que traz consigo uma série de idéias, conceitos e resultados de muitas áreas da ciência como a inteligência artificial, ciência da computação, sociologia, economia, filosofia e as ciências administrativas. Segundo este estudo, os sistemas multiagentes podem ser usados em várias aplicações, inclusive em sistemas de manufatura.

Lim (1999), propõe um novo esquema na aplicação da tecnologia dos agentes no processo dinâmico de planejamento e escalonamento, que deve resultar no aumento da resposta dos sistemas de manufatura. Neste esquema, um grupo ótimo de máquinas é escolhido com a finalidade de produção de determinado bem, baseado em critérios de custos, datas, confiabilidade das máquinas e gargalos. A adequação dos grupos de máquinas facilita a utilização otimizada dos recursos de manufatura tão bem quanto uma avaliação prévia de configurações alternativas destes sistemas. Um modelo de sistemas de manufatura é usado neste esquema para representar o que está sendo planejado, auxiliando, deste modo, a avaliação das opções dos planos de processo e escalonamento gerados.

Marík (1998) apresenta uma pesquisa focada principalmente na chamada produção orientada a projeto onde se pode somente ver um único ou limitado número de peças de um dado produto para ser produzido. Neste estudo, Marík implementou um protótipo de um sistema chamado ProPlanT (PROduction PLANning Technology), onde é mantida uma família de agentes que foram projetados para simular todo o processo de planejamento de produção.

Pereira (1999) apresenta um esquema que combina objetos distribuídos de tempo real com algoritmos adaptativos de escalonamento que suportam modos e mudanças de modos para o desenvolvimento de sistemas multiagente em aplicações industriais de tempo real. O domínio de aplicação dos agentes nesta abordagem é a flexibilidade e adaptabilidade para plantas de manufatura, incluindo recursos que devem

ter sua utilização otimizada em face de mudanças de estratégias e prioridades de produção, manutenção e emergência.

O trabalho de Balduzzi (1999) apresenta uma abordagem alternativa para o problema de estratégias de controle para sistemas de manufatura baseada no princípio de dividir e conquistar, observando que um sistema físico complexo é composto por um conjunto de sub-sistemas interconectados; e, o comportamento de todo o sistema aparece a partir das interconexões entre os sub-sistemas. Assim, é possível obter comportamentos desejados a todo o sistema impondo um comportamento adequado a cada um dos sub-sistemas. O objetivo deste trabalho é descrever uma abordagem multiagente para descentralizar o controle em sistemas de manufatura através de modelos matemáticos.

Um novo conceito de diagnóstico automático baseado em uma arquitetura multiagente aberta, robusta e flexível para o diagnóstico de processos industriais é proposto por Sanz-Bobi (1999). Para alcançar este objetivo, um projeto de pesquisa chamado DIAMOND (Distributed Architecture for Monitoring and Diagnostics), foi proposto, focando a integração de vários sistemas de monitoração e diagnóstico simples em um sistema global, permitindo que estes cooperem entre si.

2.2 Sistemas Holônicos de Manufatura

Em 1967, o autor e filósofo húngaro Arthur Koestler propôs a palavra “Holon” para descrever uma unidade básica de organização em sistemas biológicos e sociais (KOESTLER, 1967). A palavra Holon é a combinação da palavra grega “holos”, que significa “um todo” e o sufixo “on”, que significa partícula ou parte. Um holon, conforme Koestler imaginou, é uma parte identificável de um sistema que tem uma identidade única, sendo composto de partes menores e que por sua vez é parte de um todo maior (HOLONIC MANUFACTURING SYSTEMS, 2002).

A força da organização holônica (holarquia) é a de que se permite a construção de sistemas muito complexos que são eficientes no uso dos recursos, altamente resistentes a distúrbios (tanto internamente quanto externamente), e adaptativos a mudanças no ambiente em que os mesmos existem. Todas estas características podem ser observadas em sistemas biológicos ou sociais.

A teoria dos sistemas holônicos segue em grande parte os conceitos de agentes e dos sistemas multiagente. Vários pesquisadores estão engajados na criação de sistemas de manufatura ágeis a partir desta tecnologia. Conforme Bertotto (2000), vários pesquisadores têm realizado estudos que viabilizam a utilização destas idéias em sistemas de manufatura, que são citados abaixo:

A abordagem desenvolvida por Silva (1998) propõe um algoritmo de escalonamento dinâmico para arquiteturas holônicas. Esta abordagem é composta por uma holarquia de muitos holons de alto nível (processo, planejamento, etc.), formados por outros holons. Em um segundo nível, aparecem os holons de recursos, produtos e tarefas que, ao mesmo tempo, são parte dos muitos holons de alto nível, formando assim uma estrutura altamente flexível e reativa. Esta estrutura é claramente mais flexível do que as tradicionais arquiteturas CIM estáticas. No entanto, um grande esforço deve ser usado para a coordenação entre os holons para manter a mesma coerência do sistema.

A proposta apresentada por Langer (1998), consiste em um controle de chão de fábrica que permite a criação de sistemas ágeis de manufatura. Esta proposta emprega a

arquitetura denominada HoMuCS (Holonic Multi-cell Control System), baseada na arquitetura de referência dos sistemas holônicos de manufatura (HMS). Conforme os autores, é impraticável construir um controle de chão de fábrica baseado na arquitetura de referência. Portanto, há a necessidade de uma arquitetura e de uma metodologia correspondente, que é dedicada ao desenvolvimento e a implementação de sistemas de controle de chão de fábrica. A arquitetura HoMuCS é caracterizada por uma abordagem *top-down*. Inicialmente especifica-se as relações estruturais entre os holons, descrevendo-se, então, a estrutura e a funcionalidade de cada holon.

2.3 Orientação a Objetos

Segundo Rumbaugh (1994), sob um ponto de vista superficial, a expressão **orientado a objetos** significa que o software é organizado como uma coleção de objetos separados que incorporam tanto a estrutura quanto o comportamento dos dados. Isso contrasta com a programação convencional, segundo a qual a estrutura e o comportamento dos dados têm pouca vinculação entre si. Existe alguma discordância sobre quais são exatamente as características exigidas pela abordagem baseada em objetos, mas elas geralmente incluem quatro aspectos: identidade, classificação, polimorfismo e herança.

Identidade significa que os dados são subdivididos em entidades discretas e distintas, denominadas **objetos**. Cada objeto tem sua própria identidade, que lhe é inerente. Em outras palavras, dois objetos são distintos mesmo que todos os valores de seus atributos (como nome e tamanho) sejam idênticos.

No mundo real, um objeto limita-se a existir, mas, no que se refere a uma linguagem de programação, cada objeto dispõe de um único **indicador**, pelo qual ele pode ser referenciado inequivocamente. O indicador pode ser implementado de diversas maneiras, como um endereço, um elemento de uma matriz ou um valor exclusivo de um atributo. As referências dos objetos são uniformes e independentes do conteúdo dos mesmos, permitindo a criação de coleções de objetos mesclados, tal como um diretório de um sistema de arquivos que contenha tanto arquivos quanto subdiretórios.

Classificação significa que os objetos com a mesma estrutura de dados (atributos) e o mesmo comportamento (operações) são agrupados em uma classe. Uma **classe** é uma abstração que descreve propriedades importantes para uma aplicação e ignora o restante. Qualquer escolha de classe é arbitrária e depende da aplicação.

Cada classe descreve um conjunto possivelmente infinito de objetos individuais. Cada objeto é dito ser uma **instância** de classe. Cada instância de classe tem seu próprio valor para cada atributo, mas compartilha os nomes de atributos e operações com outras instâncias da mesma classe.

Polimorfismo significa que a mesma operação pode atuar de modos diversos em classes diferentes. Uma **operação** é uma ação ou transformação que um objeto executa ou a que ele está sujeito. Uma implementação específica de uma operação por uma determinada classe é chamada **método**. Como um operador baseado em objeto é polimórfico, pode haver mais de um método para sua implementação.

No mundo real, uma operação é simplesmente uma abstração de um comportamento análogo entre diferentes tipos de objetos. Cada objeto sabe como executar suas próprias operações. Entretanto, uma linguagem de programação baseada em objetos seleciona automaticamente o método para implementar uma operação com

base no nome da operação e na classe do objeto que esteja sendo operado. O usuário de uma operação não necessita saber quantos método existem para implementar uma determinada operação polimórfica, pois novas classes podem ser adicionadas sem que se modifique o código existente. São fornecidos métodos para cada operação aplicável em novas classes.

Herança é o compartilhamento de atributos e operações entre classes com base em um relacionamento hierárquico. Uma classe pode ser definida de forma abrangente e depois refinada em sucessivas subclasses mais definidas. Cada subclasse incorpora, ou **herda**, todas as propriedades de sua superclasse e acrescenta suas próprias e exclusivas características. As propriedades da superclasse não precisam ser repetidas em cada subclasse. A capacidade de identificar propriedades comuns a várias classes de uma superclasse comum e de fazê-las herdar as propriedades da superclasse pode reduzir substancialmente as repetições nos projetos e programas e é uma das principais vantagens dos sistemas baseados em objetos.

Para Booch (1994), um modelo baseado em objetos deve possuir sete propriedades, sendo quatro fundamentais (abstração, encapsulamento, modularização e hierarquia), e três complementares (tipagem, concorrência e persistência).

A **abstração** é uma simplificação que enfatiza os aspectos mais importantes e negligencia os demais.

O **encapsulamento** é um conceito complementar ao da abstração. A abstração se preocupa com o comportamento observável de um objeto, enquanto que o encapsulamento focaliza o suporte que dará a manutenção deste comportamento. O encapsulamento gerencia a complexidade na medida que somente as informações realmente necessárias a outros módulos são feitas visíveis na interface dos módulos.

A **modularização** é o ato de particionar um programa em elementos individuais, fracamente acoplados, com o objetivo de reduzir a sua complexidade criando um conjunto de limites bem definidos. Módulos servem de repositórios onde se armazenam classes e objetos relacionados.

Hierarquia é a forma de relacionamento entre as classes de um programa. Os dois tipos de hierarquias mais importantes 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” e “é parte de”.

Tipagem é a propriedade que a metodologia possui de formar modelos de referência. Na orientação a objetos cada classe define um tipo diferente que pode ser instanciado na forma de um ou mais objetos.

A **concorrência** ocorre quando partes de um mesmo programa são executadas simultaneamente em fluxos de execução distintos. Os objetos são unidades naturais para a execução concorrente, permitindo que aplicações do mundo real com concorrência intrínseca sejam modeladas naturalmente.

A **persistência** é a propriedade de um objeto cuja existência transcende tempo e/ou espaço. 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.

2.4 Simulação

Conforme Banks (1995), uma simulação é uma imitação da operação de um sistema ou processo do mundo real no decorrer do tempo. A simulação envolve a geração de uma história artificial e a observação desta história para obter inferências sobre as características de operação do sistema real.

Para Law (1991), simulação é uma técnica onde as operações de vários tipos de processos e recursos do mundo real são imitados ou simulados. Os processos ou recursos de interesse são chamados de sistema. A fim de estudar o sistema de modo científico, deve-se criar um conjunto de suposições sobre como o mesmo funciona. Estas suposições, que usualmente têm a forma de relações lógicas e matemáticas, constituem o modelo, que é usado para entender como o sistema correspondente comporta-se.

Law cita algumas áreas de aplicação da simulação. Abaixo está uma lista de alguns problemas particulares na qual a simulação se mostrou uma ferramenta poderosa:

- Projetar e analisar sistemas de manufatura;
- Avaliar requisitos de hardware e software para sistemas de computação;
- Avaliar novas armas militares e táticas de guerra;
- Determinar políticas de inventário;
- Desenvolver sistemas de comunicação e protocolos de mensagens para aqueles;
- Projetar e operar sistemas de transporte como trens, metrôs, aeroportos, rodovias e portos;
- Avaliar projetos de serviços como hospitais, correios ou restaurantes;
- Analisar sistemas econômicos e financeiros;

O comportamento do sistema em relação ao tempo é estudado com auxílio de um modelo de simulação. Este modelo usualmente tem a forma de um conjunto de hipóteses a respeito de sua operação. Estas hipóteses são expressas em relações matemáticas, lógicas e simbólicas entre as entidades ou objetos de interesse do sistema. Uma vez desenvolvido e validado, o modelo pode ser usado para investigar uma grande variedade de questões sobre o sistema real. Mudanças potenciais do sistema real podem ser simuladas antes de modo a prever o impacto no seu desempenho. A simulação pode ser usada também para estudar sistemas em tempo de projeto, antes de este ser construído. Deste modo, a simulação pode ser usada tanto como uma ferramenta de análise para avaliar os efeitos das mudanças no sistema existente, quanto como ferramenta de projeto para prever a performance dos novos sistemas sobre um vasto conjunto de circunstâncias.

Em algumas instâncias, um modelo pode ser simples o suficiente para ser avaliado por métodos matemáticos. No entanto, muitos sistemas reais são tão complexos que os modelos destes sistemas são virtualmente impossíveis de serem solucionados através de métodos matemáticos. Neste caso, a simulação numérica baseada em computadores pode ser usada para imitar o comportamento do sistema em relação ao tempo.

A disponibilidade de linguagens de simulação, capacidade de computação massiva e os avanços na metodologia de simulação têm transformado a simulação em uma das ferramentas de análise e investigação mais largamente usadas e aceitas pela comunidade científica.

Para modelar um sistema, é necessário entender o conceito de um sistema e seu limite. Um sistema é definido como um grupo de objetos que são ligados por alguma interação regular ou alguma interdependência buscando a realização de algum objetivo. Um sistema é freqüentemente afetado por mudanças fora do sistema. Estas mudanças ocorrem no contexto do sistema. Na modelagem de sistemas, é necessário decidir os limites entre o sistema e seu ambiente. Esta decisão depende da finalidade do estudo realizado.

Algumas vezes é interessante estudar o modelo para entender a relação entre seus componentes ou prever como o sistema irá operar em sua nova política. Outras vezes, é possível experimentar o sistema utilizando-o. No entanto, nem sempre isto é possível. Um novo sistema pode não existir; pode ser apenas um sistema na forma hipotética ou no estágio de projeto. Mesmo se o sistema existir, pode ser impraticável estudá-lo. Conseqüentemente, os estudos dos sistemas são realizados com um modelo do sistema.

Um modelo é definido como uma representação do sistema com a finalidade de estudá-lo. Para a maioria dos estudos, somente é necessário considerar os aspectos que afetam o sistema em investigação. Estes aspectos são representados em um modelo do sistema e este modelo, por definição, é uma simplificação do sistema. Por outro lado, o modelo deve ser suficientemente detalhado para permitir que conclusões válidas possam ser tecidas sobre o sistema real. Modelos diferentes do mesmo sistema podem ser requeridos para investigar mudanças.

Para Law, há alguns impedimentos para a grande aceitação da simulação. Primeiro, os modelos usados para estudo de sistemas em grande escala tendem a ser complexos e escrever programas de computador para executá-los pode ser uma tarefa árdua. Esta tarefa tem sido facilitada nos últimos anos com o desenvolvimento de excelentes programas que automaticamente oferecem muitos recursos para a codificação de modelos de simulação. Um segundo problema na simulação de sistemas complexos é o vasto tempo computacional necessário para a execução de uma simulação. Entretanto, esta dificuldade está tornando-se menos severa à medida que os recursos computacionais tornam-se cada vez mais poderosos e baratos.

2.5 Simulação em Sistemas de Manufatura

Banks argumenta que os sistemas de manufatura são uma das aplicações mais importantes da simulação. A simulação tem sido usada com sucesso no auxílio do projeto de novas instalações de produção, depósitos e centros de distribuição. Tem sido usada também para avaliar melhorias sugeridas aos sistemas existentes. Os engenheiros e analistas que usam a simulação têm encontrado seu valor para avaliar o impacto do investimento de capital em equipamentos e instalações físicas, e mudanças propostas para movimentação de materiais e leiaute físico. Também é útil para avaliar recursos humanos e regras de operação, e propor regras e algoritmos para serem incorporados em sistemas de gerenciamento de depósitos. Os gerentes e diretores têm encontrado na simulação uma maneira de testar o sistema antes de fazer investimentos de capital e antes de interromper o sistema atual com mudanças não provadas.

Law pondera sobre o dramático aumento no uso da simulação para o projeto e otimização de sistemas de manufatura e locais de depósito. Algumas razões para este aumento incluem:

- O aumento da competição entre as indústrias tem resultado em uma maior ênfase na automação para aumentar a produtividade e qualidade dos produtos, reduzindo seus custos;
- O custo dos sistemas computacionais tem sido reduzido com a introdução de microcomputadores e estações de trabalho;
- Melhorias constantes nas ferramentas computacionais de simulação têm reduzido o tempo de desenvolvimento de modelos, permitindo uma melhor análise dos sistemas em questão;
- A viabilidade da animação tem resultado em um melhor entendimento do uso da simulação pelos analistas.

Para ser efetivo e não perder a identificação dos problemas reais, a simulação de sistemas de manufatura, como em todos os modelos, precisam conter o nível correto de detalhe. As diretrizes mais importantes para capturar o correto nível de detalhamento são os objetivos do estudo e as questões que devem ser inquiridas. O nível de detalhe é restringido pela disponibilidade de dados de entrada e do conhecimento de como os componentes do sistema operam. Para sistemas que ainda não existem, a disponibilidade de dados podem ser limitadas e as características dos sistemas são apenas baseadas em suposições. Estas diretrizes somadas à experiência dos analistas proporcionam o embasamento para alcançar o correto nível de detalhe.

2.5.1 Objetivos da Simulação em Sistemas de Manufatura

Para Banks e Law, o maior benefício da simulação vem da compreensão e do entendimento obtido das operações do sistema. A visualização da simulação com ajuda de animações e gráficos proporciona um melhor auxílio na comunicação das suposições, operações do sistema e resultado do modelo. Freqüentemente, a visualização é a parte que mais contribui para a credibilidade do modelo, o que por conseqüência, melhora a aceitação dos resultados numéricos do modelo. É evidente que um projeto adequado que inclui a faixa correta de condições experimentais, somado a uma rigorosa análise e, para modelos estocásticos, uma análise estatística apropriada, é o fator mais importante para que conclusões corretas sejam alcançadas na saída da simulação.

Os maiores objetivos da simulação em sistemas de manufatura são os de identificar áreas problemáticas e quantificar a performance do sistema. Algumas das medidas mais comuns da performance do sistema incluem:

- Rendimento do sistema em cargas médias e máximas;
- Utilização de recursos, máquinas e mão-de-obra;
- Pontos de gargalo e obstruções;
- Filas;
- Atrasos ocorridos por sistemas e dispositivos de movimentação de materiais;

- Mão-de-obra necessária;
- Eficiência do escalonamento de produção;
- Eficiência dos sistemas de controle;
- Aumento da produção do sistema (produtos produzidos por unidade de tempo);
- Redução dos inventários entre atividades;
- Aumento da utilização de máquinas e mão-de-obra;
- Diminuição do tempo de entrega de pedidos aos clientes;
- Redução do capital de investimento necessário;
- Redução dos custos de operação;
- Certeza sobre o quanto o sistema irá, de fato, produzir;
- As informações obtidas para a construção do modelo de simulação irão promover um maior entendimento do sistema, que acaba acarretando outros benefícios;
- A construção do modelo de simulação para o sistema proposto obriga os projetistas a pensarem sobre certos problemas por mais tempo do que normalmente pensariam.

A simulação alerta com sucesso os projetistas quanto a problemas particulares em sistemas de manufatura, que podem ser classificados em três categorias gerais:

- A necessidade de equipamentos e recursos humanos:
    - Número e tipo de máquinas para um objetivo particular;
    - Número, tipo e arranjo físico de equipamentos de suporte e transporte (esteiras, empilhadeiras, AGV's);
    - Localização e tamanho de depósitos de inventário;
    - Avaliação dos efeitos no sistema com a adição de um novo equipamento;
    - Avaliação da mudança de volume ou mix de produtos;
    - Planejamento de recursos humanos necessários;
- Avaliação de Performance
    - Análise de produção;
    - Análise de tempo de produção;
    - Análise de gargalos.

- Avaliação de Procedimentos Operacionais
    - Escalonamento de produção;
    - Políticas para organização de partes e materiais de estoque;
    - Estratégias de controle;

- Análise de confiabilidade;
- Políticas de controle de qualidade.

Há várias medidas comuns de performance obtidas a partir de estudos de simulação em sistemas de manufatura, incluindo:

- Capacidade de produção;
- Tempo das partes no sistema;
- Tempo das partes em filas;
- Tempos de espera por recursos de transporte;
- Oportunidades de entregas;
- Tamanho dos estoques entre processos;
- Utilização de equipamentos e recursos humanos;
- Proporções de tempo em que um recurso está indisponível, avariado, bloqueado ou em manutenção preventiva;
- Proporções de peças que necessitam de retrabalho ou são refugadas.

2.5.2 Ambientes de Simulação para Sistemas de Manufatura

A simulação de sistemas de manufatura e movimentação de materiais é tão complexa que ferramentas especiais de simulação foram desenvolvidas. Existem vários ambientes de simulação no mercado para este propósito. São apresentados a seguir alguns dos mais importantes ambientes para simulação de sistemas de manufatura disponíveis no mercado:

2.5.2.1 SimFactory II.5

SIMFACTORY II.5 (BANKS, 1995) é um simulador voltado para aplicações fabris escrito nas linguagens SIMSCRIPT II.5 e MODSIM III. Este simulador opera em várias plataformas computacionais como PC's e estações de trabalho SUN. O modelo é facilmente construído em estágios, através da definição do leiaute dos produtos, dos recursos, dos transportes e finalmente das interrupções. Uma animação icônica segue automaticamente as definições do modelo. O leiaute é criado posicionando-se ícones, selecionados a partir de uma biblioteca, na tela. À medida que cada ícone é posicionado, as características correspondentes vão sendo adicionadas. Os produtos são definidos por roteiros que definem as operações executadas por cada parte e duração de cada operação.

Os relatórios podem mostrar a utilização de equipamentos, produção por minuto, tempo de produção e utilização das filas. Múltiplos gráficos podem ser comparados ao mesmo tempo. Os dados podem ser comparados também em vários experimentos. Os relatórios podem ser configurados ou exportados para uma planilha eletrônica e estatísticas especiais podem ser coletadas nos elementos de interesse.

2.5.2.2 Automod

O Automod (AUTO SIMULATIONS, 2002), da empresa Auto Simulations, Inc., combina as características de uma linguagem de propósito geral com um simulador para movimentação de materiais e sistemas de manufatura. Ele possui recursos genéricos de programação incluindo a especificação de processos e procedimentos de processos, recursos, cargas, filas e variáveis. Os processos são especificados em termos de limites de tráfego, conexões de entrada e saída para sistemas de movimentação de materiais e processamento lógico. Os recursos são especificados levando em conta sua capacidade, tempo de processamento, tempo entre falhas e tempo de reparo. As cargas são definidas por seu formato e tamanho, atributos, taxas de geração, limites de geração e tempo de início, além de sua prioridade.

O simulador de movimentação de materiais é bastante poderoso. AGV's, esteiras, pontes móveis, AS/RS, dentre outros sistemas podem ser definidos. Por exemplo, um AGV pode ser definido como múltiplos veículos, veículos de múltipla capacidade, opções de caminhos, velocidade, aceleração e desaceleração baseada em tipos de carga e de caminhos, pontos de controle, controle flexível e regras de escalonamento, roteamento automático de distâncias mais curtas e procedimentos para adaptar os veículos.

Numerosas declarações de controle, chamadas ações, estão disponíveis no sistema. Controle de cargas, recursos e outras ações também estão disponíveis. Funções escritas na linguagem de programação C também podem ser chamadas, mas geralmente não são necessárias na maioria dos sistemas. Atributos e variáveis podem ser especificados.

As capacidades de animação são baseadas em desenhos em escala e incluem gráficos 3-D que podem ser rotacionados e aproximados em uma tela virtual em tempo real. Desenhos em formato CAD podem ser usados para construir os elementos gráficos do modelo. Gráficos de negócios, incluindo os de barra, de pizza e linhas de tempo podem ser gerados. A figura 1 mostra a interface de modelagem do Automod.

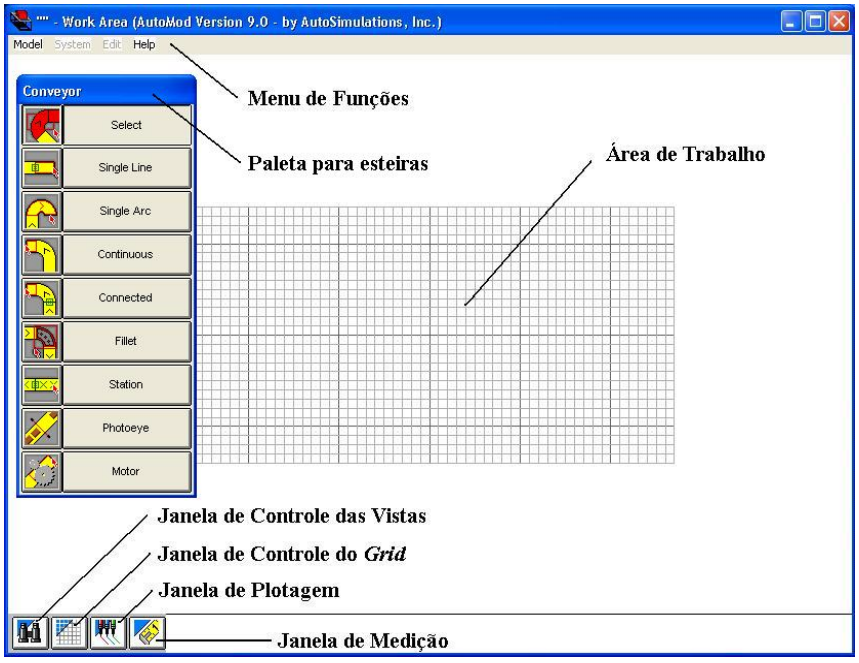

FIGURA 1 – Interface de modelagem do Automod com a paleta de componentes para construção de esteiras

Pelo fato do Automod ser usado como ferramenta de comparação neste trabalho, ele é exposto de forma mais aprofundada no capítulo 4, seção 4.2, onde são apresentadas a maioria dos passos para a criação de um modelo de simulação.

2.5.2.3 Taylor II

Taylor II (BANKS, 1995) é um produto alemão produzido pela F&H Simulations B. V. Um modelo no Taylor II consiste em quatro entidades fundamentais: elementos, tarefas, roteamentos e produtos. Os tipos de elementos são de entrada, saída, máquinas, filas, esteiras, transportes, caminhos, apoio, depósitos e estoques. As três operações básicas são: processamento, transporte e armazenagem.

A definição do leiaute, que consiste de tipos de elementos, é o primeiro passo quando se constrói um modelo. Selecionando os elementos em seqüência, o caminho ou rota do produto é definido. As descrições do roteamento podem ser descritas por arquivos externos.

O próximo passo é o detalhamento do modelo. Inicialmente são inseridos os parâmetros. Em adição ao número de valores padrão, o Taylor II usa uma linguagem de macro chamada TLI (Taylor Language Interface), a qual permite modificações no comportamento do modelo em combinação com variáveis definidas pelo usuário. Além disso, esta linguagem pode ser usada interativamente durante a execução da simulação para permitir atualizações. Também estão disponíveis interfaces para linguagens de programação como C, Pascal e Basic.

Durante a simulação são possíveis ações como aproximação, rotação e pausa. Modificações podem ser feitas em tempo de execução. A representação do tempo é definida pelo usuário.

As possibilidades de análise de saída incluem gráficos pré-definidos, definidos pelo usuário, relatórios pré-definidos e definidos pelo usuário.

2.5.2.4 Witness

WITNESS (BANKS, 1995), da empresa AT&T Istel, contém muitos elementos para a simulação discreta de sistema de manufatura e possui uma orientação explícita a tipagem de máquinas conforme suas funções. Pode-se especificar variáveis e atributos e escalonar as peças que chegam ao sistema através de arquivos externos. Este software possui vários tipos de distribuições estatísticas que podem ser usadas a qualquer momento na simulação.

As ações da simulação, executadas no início e no final dos eventos simulados, podem usar comandos da linguagem de programação C-LINKS, que permite adicionar programas detalhados e sub-rotinas complexas. Inclui também a capacidade de produzir relatórios sobre dados das máquinas e elementos da simulação dinamicamente na tela. Além disso, possui uma ferramenta de animação que permite a visualização dos resultados da simulação de forma gráfica.

2.5.2.5 Arena

A ferramenta Arena (BANKS, 1995), da empresa Systems Modeling Corporation, é um exemplo de um pacote de animação e simulação. Sua intenção é aproveitar o poder da linguagem SIMAN. SIMAN é um mecanismo de programação e animação sobre o qual a ferramenta Arena foi construída.

Um modelo de simulação usando Arena é construído através da seleção de módulos que contém as características completas de um processo. Uma vez que todos os módulos estejam especificados e seus relacionamentos criados, o simulador pode executar modelos animados dos processos em questão.

Os módulos podem ser organizados em *templates* especializados para cada domínio de aplicação. Usando esta habilidade de customização, podem ser criados gabaritos para uma empresa ou departamento específicos que podem facilitar a construção de outras simulações por usuários sem conhecimentos profundos sobre simulação.

2.6 Simulação Orientada a Objetos em Sistemas de Manufatura

A força do paradigma de orientação a objetos aplicados a modelagem e simulação tem recebido considerável atenção dos pesquisadores. A correspondência um para um entre objetos no problema real e suas abstrações no software provido pelo paradigma de orientação a objetos, oferecem potencial para uma modelagem mais intuitiva. Neste caso, a orientação a objetos está sendo explorada para permitir a construção de softwares de simulação fiéis ao modelo de modo a suportar a modelagem e controle de sistemas de manufatura complexos e integrados.

Conforme Roberts (1998), a vantagem mais tipicamente citada no uso da simulação orientada a objetos é a habilidade de modelar sistemas usando entidades que são naturais ao sistema. Outras vantagens citadas incluem o rápido desenvolvimento, aumento da qualidade dos modelos, fácil manutenção, reusabilidade de projeto e software e redução do risco de desenvolvimento de sistemas complexos.

No entanto, a simulação orientada a objetos freqüentemente requer um vasto conhecimento prévio sobre os mecanismos e programas de simulação, além do fato de que a construção inicial das classes pode tomar um longo tempo.

A mais fundamental diferença entre as linguagens de simulação e a simulação orientada a objetos é de que na primeira os projetistas devem escrever programas customizados ou combinar os recursos fixos em muitos problemas, diminuindo, neste caso, o poder de modelagem. Outra diferença fundamental é a inabilidade para definir, combinar, expandir e reusar primitivas de programação. A hierarquia de classes da proposta orientada a objetos permite ao projetista acessar e manipular todos os níveis do código no ambiente do modelo.

A principal discussão é a flexibilidade da modelagem. Qualquer linguagem de simulação, baseada ou não na orientação a objetos, irá apresentar algum tipo de barreira na modelagem do problema, exceto se houverem algumas facilidades para a criação de novas entidades e interconexões entre estas. Um recurso parcial em muitas linguagens de simulação é o acesso a uma linguagem raiz para os casos onde as construções para a modelagem oferecidas são insuficientes. As linguagens orientadas a objetos têm

vantagens no que se refere a como os objetos são criados e conectados. Da perspectiva da programação, os objetos e entidades com características descritivas e capacidades funcionais são encapsulados em uma unidade auto contida. Os mecanismos básicos para a criação de novos objetos e suas interfaces são características herdadas das linguagens de programação orientadas a objeto.

Com este mecanismo, o projetista não fica limitado a uma pré-definição de conjuntos de entidades dos modelos e pode criar entidades que se aproximem da cognição humana. A orientação a objetos oferece ao projetista a possibilidade de construir modelos hierárquicos através da decomposição do modelo em componentes lógicos, ou objetos, a partir do ponto de vista do projetista. Objetos que possuem fronteiras físicas naturais são freqüentemente o resultado desta capacidade.

As classes são organizadas em hierarquias que proporcionam uma estrutura para acomodar hierarquias de sub-modelos. Podem ser estruturadas hierarquicamente tanto as definições dos objetos quanto seus comportamentos. Este conceito reflete uma abordagem estruturada para a composição do modelo no qual os objetos em um nível de abstração podem ativar objetos em outros níveis mais baixos. Esta abordagem *top-down* permite ao projetista facilmente modelar sistemas reais em qualquer nível de detalhe. A orientação a objetos é **extensível** porque o projetista pode definir novos objetos com comportamentos especiais a partir de classes, objetos e métodos disponíveis. As definições das classes também permitem e facilitam o **reuso** dos componentes modelados.

Um exemplo usando uma simples máquina pode ser usado para demonstrar as diferenças entre a modelagem de simulação orientada a objetos e a modelagem tradicional, voltada para linguagens de simulação orientadas a processos. Neste exemplo, as peças chegam ao sistema randomicamente, onde são processadas pela máquina durante um tempo especificado. Se a máquina não está disponível, as peças esperam em uma fila até que a máquina torne-se disponível novamente. Quando o processo se completa, as partes deixam o sistema.

Na abordagem orientada a processos, a modelagem é centrada no fluxo de entidades. As propriedades e características das entidades são guardadas como atributos. Por exemplo, o tempo em que a peça entra no sistema é representado como um atributo. Este atributo pode ser usado depois para computar o ciclo da peça. A figura 2 descreve a modelagem lógica de uma linguagem de simulação orientada a processos, onde os retângulos podem representar blocos ou nodos. Um bloco/nodo especial é responsável pela criação randômica das peças. As peças estão fluem para o bloco/nodo, onde irá esperar até que o recurso correspondente esteja disponível. Uma vez que a máquina fique disponível, a entidade (peça) é atendida pela mesma durante um tempo chamado tempo de processamento da máquina. Após, a peça é passada para outro bloco/nodo representando que a peça foi liberada pela máquina.

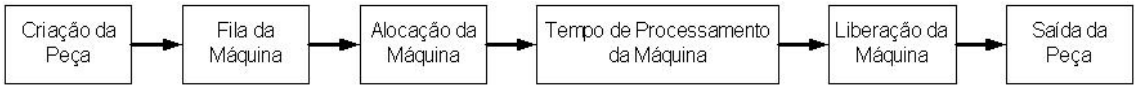

FIGURA 2 – Modelagem lógica para uma linguagem orientada a processos

Na abordagem orientada a objetos, os objetos, pertencentes a classes interagem ou comunicam-se através de passagem de mensagens. Por exemplo, algumas classes

necessárias podem ser uma Fila de Eventos, um Gerador de Números Randômicos, as Peças, Máquinas e a Fila. Além disso, cada objeto, independente do seu tipo, pode ter seus próprios atributos. Este é o contraste com linguagens de simulação tradicionais, onde somente as entidades podem ter propriedades ou características associadas a ela. A figura 3 apresenta uma possibilidade de cenário de uma modelagem orientada a objetos para o exemplo da máquina. Os círculos representam as classes e as flechas descrevem a passagem de mensagens entre os objetos.

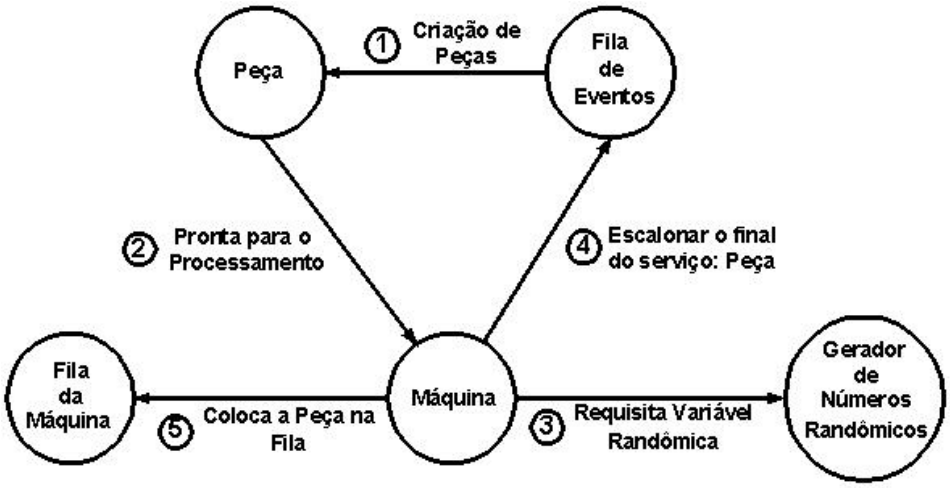

FIGURA 3 – Exemplo da modelagem orientada a objetos

O objeto fila de eventos manda uma mensagem para o objeto peça a fim de criá-lo. Após isto, a peça envia uma mensagem para a máquina indicando que a mesma está pronta para ser processada. O objeto máquina deve então ativar um método para checar se a mesma está ou não ocupada. Caso negativo, uma requisição é feita ao objeto gerador de números randômicos para criar um tempo de processamento gerado a partir de uma distribuição especificada. Ao final do atendimento do objeto peça e máquina é então enviada uma mensagem para o objeto Lista de Eventos. Caso a máquina esteja ocupada, o objeto peça é escalonado no objeto fila correspondente ao objeto máquina.

Segundo Govindaraj (1990), a questão crítica no desenvolvimento de software de simulação é o projeto das classes de objetos genéricas adequadas. Tem-se adotado alguns princípios no projeto da hierarquia de classes de objetos para sistemas de manufatura. O problema fundamental na simulação se refere à modelagem, isto é, usar as abstrações de construção nas linguagens de simulação para descrever o sistema a ser analisado. Todos os modelos de simulação representam uma abstração da realidade. A construção do modelo de simulação usando uma linguagem particular é limitada pela estrutura desta linguagem.

A chave da questão na exploração da orientação a objetos para a simulação de sistemas de manufatura é a identificação da classe de objetos genérica e seu conteúdo. Os princípios de construção utilizados para criar esta hierarquia podem ser categorizados em princípios específicos para manufatura e princípios gerais de orientação a objetos. Narayanan (1992) comprova o que foi citado acima propondo alguns princípios específicos para o desenvolvimento de hierarquia de classes para sistemas de manufatura que facilitam a sua construção.

Em seu estudo, Narayanan (1998) e Bathaglini (1997) citam um subconjunto de esforços de pesquisa mais relevantes no sentido da especificação de hierarquias de classes orientadas a objetos para a aplicação em simulação de sistemas de manufatura, que são resumidos nas seções a seguir:

2.6.1 BLOCS/M

Criado na Universidade da Califórnia, Berkeley. BLOCS/M foi a primeira arquitetura orientada a objetos específica para aplicações em simulação de sistemas de manufatura, aplicada preferencialmente em problemas da indústria de semicondutores. Possui quatro abstrações fundamentais para representar objetos de manufatura: lotes, recursos, tarefas e rotas. BLOCS/M é focado no desenvolvimento de bibliotecas de classes reutilizáveis, que são projetadas de acordo com o princípio de que cada objeto realiza apenas uma tarefa. BLOCS/M usa sub-classes para desenvolver objetos que apresentam novos tipos de comportamento a partir das classes existentes.

A especificação do comportamento em BLOCS/M é composta por eventos que são representados pelos métodos das classes. Além disso, BLOCS/M possui um objeto chamado tarefa (task) que essencialmente aloca e desaloca os objetos de recurso durante a simulação.

O acoplamento da arquitetura BLOCS/M é obtido através da construção de objetos de tal simplicidade que não imponham restrições em sua construção em uma simulação específica. Os objetos são declarados, mas os relacionamentos são feitos para cada aplicação.

2.6.2 DEVS

Desenvolvido na Universidade do Arizona, DEVS é tanto uma arquitetura quanto uma implementação de software, com aplicações em sistemas autônomos de modelagem. As abstrações fundamentais neste esquema são os modelos atômicos e os componentes construídos a partir destes modelos atômicos. Não há abstrações que possam ser mapeadas diretamente em sistemas físicos. Estes sistemas devem ser construídos a partir de um conjunto de subsistemas independentes.

O acoplamento entre subsistemas é feito por meio de interfaces padronizadas, interagindo por sobre portas de entrada e saída, através de troca de mensagens. As transições internas e externas, saídas e os mecanismos de avanço do tempo são especificados para todos os subsistemas no modelo. Estas especificações caracterizam o comportamento das entidades simuladas.

2.6.3 Laval

Criado na Universidade Laval, esta arquitetura é focada nas várias fases do projeto nas indústrias, incluindo a simulação. A visão de um sistema de manufatura nesta abordagem é a de que os sistemas são organismos compostos por entidades inteligentes denominadas agentes e entidades não-inteligentes denominadas objetos. Os agentes executam uma variedade muito grande de papéis nos sistemas de manufatura e são responsáveis por interagir com outros agentes, tomando decisões para o sistema.

A definição comportamental de Laval utiliza métodos nos organismos para modelar o comportamento das entidades que estes organismos representam. Estes métodos são chamados de eventos. Os agentes representam o comportamento de tomada de decisão no sistema e os relacionamentos entre os agentes são especificados durante a geração do modelo.

O acoplamento em Laval também se baseia em interfaces padrão. O inter-relacionamento entre os agentes é programado com ajuda de hipergrafos, que são abstrações especializadas que representam interações.

2.6.4 OOSIM

Pesquisado no Instituto de Tecnologia da Geórgia, o OOSIM é focado na operação autônoma, bem como no controle supervisório humano em sistemas de manufatura integrados. As classes OOSIM são baseadas nos sistemas de manufatura discretos. Possui quatro abstrações fundamentais: materiais, locais, controladores e planos de processo. Um plano de processo especifica que operações devem ser executadas para transformar a matéria-prima em produto final. Os locais podem ser locais de processamento de materiais, locais de armazenagem ou locais de transporte e são organizados em domínios controlados. Cada domínio controlado possui um controlador, que são entidades orientadas a evento e respondem a mudanças de estado.

As interações entre domínios controlados ocorrem através de locais compartilhados. Estes locais são abstrações que permitem transferir materiais de um local para outro e são sincronizados através de eventos. Os eventos são associados com as classes. As seqüências de eventos que caracterizam o comportamento de uma entidade são encapsuladas em um modelo chamado *script*, que é baseado em grafos de eventos.

2.6.5 OSU-CIM

Desenvolvido na Universidade do Estado de Oklahoma, o OSU-CIM é focado na modelagem e simulação de peças discretas de sistemas de manufatura, com ênfase na separação do processo de modelagem das atividades de solução do problema.

Nesta abordagem, as entidades de manufatura possuem propriedades físicas, propriedades de controle e propriedades de informação. Assim como na abordagem BLOCS/M, esta arquitetura enfatiza o princípio de que cada objeto executa uma função, com o objetivo de aumentar a reusabilidade.

2.6.6 SmartSim/SmarterSim

Criado na Universidade de Michigan, Dearborn, SmartSim é focado na geração de programas de simulação para manufatura e utiliza a aplicação do formalismo DEVS para desenvolver representações que correspondem aproximadamente a entidades de manufatura. SmartSim/SmarterSim agrega abstrações fundamentais de subsistemas DEVS para criar abstrações fundamentais como peças, estações, esteiras e roteadores. As peças movem-se de uma estação para outra através de esteiras e utiliza os roteadores para representar decisões de controle. Esta metodologia também adiciona representações gráficas de entidades de manufatura.

Além dos estudos citados acima, outros autores propõem idéias similares àquelas apresentadas nesta dissertação, que são avaliadas a seguir:

Em Rosinha (2000), os autores dirigem-se para proposição de uma arquitetura de simulação flexível ao projeto de FMS, que visa a integração de simuladores a outras ferramentas computacionais de projeto FMS. Conforme o autor, esta arquitetura foi desenvolvida para a simulação de sistemas dinâmicos a eventos discretos, implementando características como a animação de chão de fábrica, a monitoração de variáveis de estado, a execução do controle e o controle do ritmo da simulação. Na figura 4, é apresentado o diagrama de classes modelado em UML desenvolvido neste estudo.

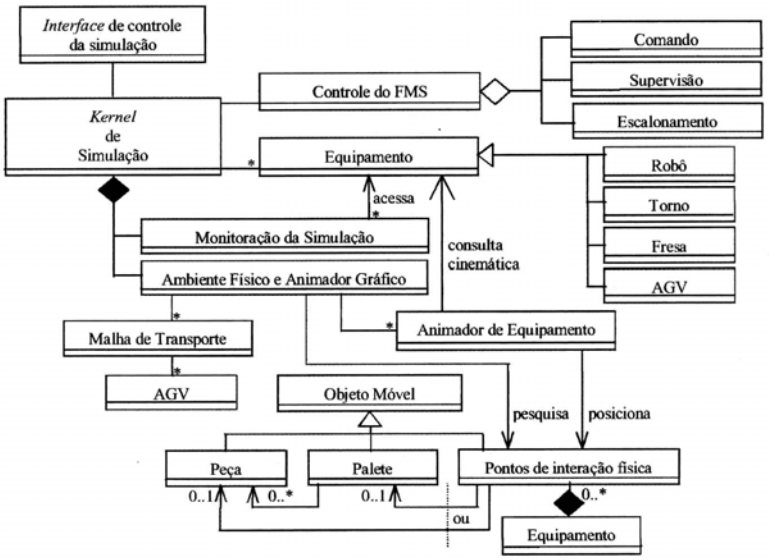

FIGURA 4 – Diagrama de Classes proposto em Rosinha

Pode-se notar que o modelo apresentado implementa funções de controle da simulação através de um Kernel de Simulação que é responsável pelas atividades de avanço de tempo da simulação e manutenção da fila de eventos. O modulo Ambiente Físico e Animador Gráfico é responsável pela animação de chão de fábrica, gerenciamento de peças e paletes, execução de interações de troca de materiais entre equipamentos e manutenção da malha de transporte.

Park (1997) introduz em seu artigo um esquema de modelagem orientado a objetos para o projeto de sistemas de manufatura automatizados (MAS) chamado JR-net. Conforme o autor, a maioria das ferramentas de modelagem existentes não são naturais para os engenheiros de automação, dificultando o aprendizado e a utilização destas ferramentas. Deve existir então uma ferramenta que seja fácil de usar e que utilize uma metodologia conhecida pelos engenheiros.

Para propor este esquema, Park baseia-se no fato de que os modernos sistemas flexíveis de manufatura são estruturas modulares e hierárquicas, construídas a partir de recursos-padrão. O trabalho é descrito de maneira formal em termos da definição dos recursos e processos, do relacionamento entre os objetos, dos procedimentos para a construção do modelo e a estrutura do modelo.

A definição dos recursos e processos foi baseada na observação dos ambientes de manufatura, resultando em recursos-padrão que são agrupados conforme suas funções. No esquema de modelagem, estes recursos são representados por símbolos que formam os nodos de uma rede de símbolos. Cada recurso é descrito internamente na forma de objetos, ou seja, a simbologia serve apenas para facilitar a visualização onde cada símbolo representa um recurso e possui uma especificação funcional na forma de classes.

O relacionamento entre os objetos é especificado através de uma hierarquia de classes onde um processo que representa uma peça desloca-se sobre uma série de estações de acordo com seu plano de processo (roteiro). Para construir o modelo, Park propõe o uso de três estágios: o leiaute estático, o modelo de deslocamento de processos e o modelo de supervisão. Estes estágios são basicamente os mesmos usados por engenheiros de automação para construir sistemas flexíveis de manufatura reais.

Segundo Park, o objetivo de seu trabalho é demonstrar que a modelagem JR-net pode ser utilizada como uma ferramenta de modelagem gráfica para simuladores comerciais, como o Automod.

2.7 CORBAManufacturing

CORBAManufacturing (OBJECT MANAGEMENT GROUP, 2002) compreende as especificações que relacionam-se com a indústria de manufatura. A principal missão de CORBAManufacturing é permitir baixo custo de produção, tempo, disponibilidade comercial e interoperabilidade em componentes de software do domínio da indústria de manufatura através da tecnologia CORBA.

CORBAManufacturing define interfaces padrão orientadas a objeto entre serviços e funções da indústria de manufatura. Estas interfaces promovem interoperabilidade entre uma variedade de plataformas, sistemas operacionais, linguagens e aplicações.

CORBAManufacturing é um domínio (CORBA Domain) da arquitetura CORBA (Common Object Request Broker Architecture). CORBA é um projeto de middleware mais importante já empreendido pela indústria. Este projeto é resultado de um consórcio de mais de 700 companhias denominado OMG (Object Management Group) (MOWBRAY, 1997) (ORFALI, 1997). O OMG Manufacturing Task Force (MfgDTF) é um grupo de trabalho da OMG que dedica-se a trazer soluções distribuídas para a indústria de manufatura.

A MfgDTF está dividida em cinco grupos distintos, listados abaixo, que se encarregam de especificar padrões de interoperabilidade CORBA em níveis específicos (OBJECT MANAGEMENT GROUP, 2002):

- **Manufacturing Common Business Objects**: definições de objetos que são comuns à indústria de manufatura e seu domínio de negócio.
- **Manufacturing Execution Systems/Machine Control**: definições de interfaces para permitir integração e operação flexível entre sistemas de computação que suportem funções de manufatura. O principal trabalho deste grupo está voltado para a especificação de padrões de aquisição de dados para sistemas industriais (Data Acquisition from Industrial Systems).

- **Product & Process Engineering**: desenvolvimento de interfaces padronizadas para engenharia de software de produtos e processos, facilitando sua produção. Este grupo trabalha em especificações para engenharia de produto e processo, interfaces CAD e gerenciamento de dados de produtos.
- **Enterprise Resource Planning**: definição de objetos para planejamento de recursos.
- **Modeling & Simulation**: desenvolvimento de interfaces para a modelagem e simulação de sistemas de manufatura. Este é o único grupo até o momento que possui uma especificação que já foi aceita pela OMG, chamada de DSS (Distributed Simulation System).

2.8 Resumo e Considerações

O mercado globalizado está exigindo mudanças nas empresas que buscam cada vez mais velocidade de produção, atendendo a maiores demandas, sem, contudo, descuidar da qualidade dos produtos. Para permitir que este panorama se concretize, novas metodologias de controle e modelagem dos sistemas de manufatura são necessárias. Os antigos paradigmas de construção destes sistemas, como a programação procedural e o controle centralizado, não são mais soluções viáveis, devido principalmente à complexidade intrínseca dos novos ambientes.

A descentralização dos sistemas de manufatura tem levado as decisões cada vez mais perto das unidades funcionais. Atualmente, os componentes destes sistemas estão sendo controlados de forma distribuída por vários recursos computacionais, que devem integrar-se para atingir o objetivo da produção. Novas pesquisas estão transferindo para os próprios componentes manufatureiros a capacidade de processamento. Assim, temos claramente a configuração de um sistema com inteligência distribuída, já que cada máquina, sensor ou atuador pode, dependendo do seu ambiente, tomar decisões e reagir sobre ele a fim de produzir algo.

Do exposto acima, podemos notar que os sistemas de manufatura podem tender para a utilização das idéias da inteligência artificial distribuída e dos sistemas multiagentes. Podemos claramente comparar um agente com um componente processado, que, devidamente programado, pode comportar-se de modo a agir de acordo com características como a reatividade, pró-atividade e habilidade social. Baseado nisso, nota-se que os sistemas multiagentes podem ser a grande solução para o problema de ambientes de manufatura ágeis que necessitam de flexibilidade e mudanças rápidas, a fim de satisfazer o atual mercado global.

Na mesma linha da inteligência artificial, as primeiras idéias propostas por Koestler (1967) sobre a teoria holônica, observam o comportamento de seres vivos em comunidades organizadas. Estas idéias foram adaptadas na criação dos Sistemas Holônicos de Manufatura após a constatação da necessidade de ambientes de manufatura altamente configuráveis e ágeis. Grande parte dos conceitos apresentados na teoria holônica são similares aos dos sistemas multiagentes, já que ambos possuem características muito semelhantes. Os sistemas holônicos de manufatura consideram a natureza distribuída dos sistemas de manufatura atuais, tentando otimizá-los com a adição de comportamentos inteligentes em seus componentes.

Como os sistemas holônicos de manufatura emprestam muitas características dos sistemas multiagentes, o nível de maturidade desta abordagem aumenta, já que a idéia

dos sistemas multiagentes existe há mais tempo. Além disso, vários grupos de pesquisa, inclusive brasileiros, estão produzindo novos trabalhos sobre este tema, provando sua viabilidade. Conforme os estudiosos da área, os sistemas holônicos, junto com a idéia de agentes inteligentes, são as alternativas mais naturais encontradas para a construção de sistemas de manufatura ágeis e altamente configuráveis. Isto é possível, pois até os menores componentes destes sistemas acompanham processamento embarcado, permitindo a eles, tomar pequenas decisões no sentido de sua execução.

Apesar do exposto acima, atualmente ainda é muito difícil a implementação destas idéias em ambientes de manufatura reais, pois as máquinas e equipamentos utilizados usam paradigmas antigos na sua construção, sendo necessárias algumas estratégias de migração dos sistemas legados e um novo tipo de controle e escalonamento. Assim, nota-se que o estado da arte desta nova tecnologia não depende somente de esforços no sentido de evoluí-la, mas também de uma integração com os fabricantes de componentes para os sistemas de manufatura, que devem construir máquinas adequadas aos novos paradigmas.

CORBA, por sua vez, é uma *middleware* que tem como grande atrativo a possibilidade de componentes inteligentes interoperarem em um barramento de objetos. Além disso, este *middleware* especifica uma gama de serviços, domínios e facilidades que auxiliam na criação e utilização de novos objetos. A tendência dos atuais sistemas de manufatura, como já comentado, tem evoluído para sistemas altamente distribuídos, adicionando-se inteligência e auto-suficiência a seus vários componentes.

CORBAManufacturing está sendo o primeiro passo para a construção de sistemas de manufatura mais fáceis de construir e gerenciar, além da possibilidade de poderem ser configuráveis conforme a aplicação. O problema é que esta tecnologia está longe de seu estado da arte. Vários fatos tornam *CORBAManufacturing* uma solução pouco atrativa para as empresas de manufatura. Um destes fatos é a pouca idade desta tecnologia. Outro ponto é a inexistência de aplicações baseadas nela, deixando para as empresas arcarem com o ônus de implementações piloto, desencorajando os investidores. Também não foram encontrados grupos de pesquisa que estejam trabalhando com estas especificações. Além de tudo isso, soma-se a demora no lançamento de novas especificações.

O que conta em favor de *CORBAManufacturing* é o apoio dado por grandes indústrias como IBM, Netscape, Oracle, Sun, Hitachi, ABB, Hewlett-Packard, Fujitsu, dentre outras. Historicamente, as grandes empresas têm o poder de impor características em projetos por elas construídos. Assim, é bem provável que em um futuro próximo, tanto a tecnologia CORBA, quanto todos seus domínios, serviços e facilidades estejam prontos e evoluindo, com plena capacidade de atingir os principais objetivos do consórcio OMG, que é o de tornar CORBA uma tecnologia padronizada de interoperabilidade de objetos distribuídos.

O que se pode concluir a partir do estudo do estado da arte na área de simulação em sistemas de manufatura é que as pesquisas estão avançando rapidamente, mas, apesar disso, os ambientes manufatureiros ainda não estão assimilando toda esta nova tecnologia, principalmente em se tratando do Brasil. As empresas ainda estão utilizando fortemente os antigos métodos de controle de manufatura e a simulação nesta área é ainda incipiente.

O exposto acima é explicado quando analisamos as diversas idéias que estão surgindo para o controle, modelagem e simulação de sistemas de manufatura, as quais estão muito dispersas. Assim, fica difícil uma empresa adotar uma tecnologia que se

encontra em um estágio inicial de desenvolvimento e que não se firmou como padrão no setor.

Dentre as idéias aqui expostas, com certeza a Orientação a Objetos é a mais promissora por se apresentar em um estágio de desenvolvimento mais avançado, se comparada com outras metodologias. Além disso, grande parte dos estudos na área de manufatura está voltada para a orientação a objetos, o que a torna uma escolha mais tangível na pesquisa de metodologias de modelagem, controle e simulação. Sem dúvida, a orientação a objetos é uma das melhores soluções para o desenvolvimento de grandes e complexos sistemas de manufatura. A partir da orientação a objetos, pode-se acoplar muitos outros esquemas de construção, pois a mesma é a peça mais básica nesta hierarquia. Para exemplificar esta afirmação, todas as outras metodologias citadas nesta análise utilizam como base a orientação a objetos. Além disso, a orientação a objetos facilita a construção de sistemas de manufatura, já que a tradução de componentes dos sistemas em classes de objetos é feita muito mais naturalmente do que na programação procedural.

Na seção 2.5.2 deste capítulo foram abordados vários ambientes de simulação de sistemas de manufatura que apresentam interfaces e estruturas de controle complexas, permitindo a construção de modelos que conseguem reproduzir fielmente uma determinada realidade. Entretanto, como será mostrado no decorrer deste trabalho, a construção de uma simulação nestes ambientes requer projetistas especializados, já que estes ambientes possuem características bastante particulares, fazendo com que o tempo de desenvolvimento seja muito longo se comparado com outras metodologias. Além disso, cada novo modelo deve ser construído praticamente do início, já que o conceito de reuso nestes casos não é implementado e as ferramentas que permitem juntar funcionalidades de vários modelos são bastante restritas. A conclusão mais clara do que foi exposto acima está focada na necessidade de um ambiente que permita o rápido desenvolvimento de modelos e incentive o reuso de estruturas já criadas, aumentando a produtividade na simulação. O estudo de vários esforços neste sentido apontou a orientação a objetos como a metodologia mais adequada, sendo usada neste trabalho para a criação de uma biblioteca de classes que permite reunir as características positivas dos simuladores comerciais com a possibilidade da rápida prototipação e do reuso, capacitando a utilização da simulação em aplicações onde não era utilizada.

3 Proposta de uma Biblioteca de Classes para Modelagem e Simulação em Sistemas de Manufatura

3.1 Introdução

O desenvolvimento da biblioteca de classes apresentada é uma evolução de trabalhos executados durante grande parte do período do curso de mestrado. As idéias contidas nesta biblioteca são melhoramentos sistemáticos de estudos feitos em trabalhos anteriores como os trabalhos individuais, artigos em congressos e experiência do autor com ambientes reais de manufatura, onde esta biblioteca foi validada. O objetivo principal desta biblioteca é facilitar a construção de modelos computacionais para simulação e análise de sistemas de manufatura.

Para permitir que esta biblioteca seja usada em linguagens de programação orientadas a objeto, uma classe adicional deve ser implementada, mas não consta no atual modelo porque foge do escopo da proposta deste trabalho. Esta classe tem como função implementar a infra-estrutura de simulação do modelo, controlando o tempo, eventos, distribuições e a possibilidade de interação externa na simulação. Ela deve implementar um mecanismo que permita a utilização ou não dos serviços de controle de simulação. Isto é necessário, pois se pode utilizar esta biblioteca também em ambientes de simulação como o SIMOO (COPSTEIN, 1997) ou sua variação para tempo real SIMOO-RT (BECKER, 1999), que já possuem o algoritmo e os serviços de controle de simulação.

O serviço de simulação deve possuir um relógio que contabiliza o tempo da simulação. Este serviço permite a interação com o usuário a fim de que o mesmo possua controle sobre a simulação. Este controle torna-se importante quando é necessário parar, aumentar ou diminuir o tempo da simulação. A única classe que se comunica com o relógio é a classe eventos futuros.

Outra função necessária do serviço de simulação é a implementação de listas de eventos futuros, além da geração de distribuições estatísticas que irão ser úteis tanto na obtenção dos resultados quanto na geração de eventos que seguem algum padrão estatístico pré-definido.

A principal característica desta biblioteca é sua abordagem que difere da maioria dos estudos correlatos na área. Neste trabalho, a abordagem usada oferece ao projetista um método de projeto similar àquele usado em seu local de trabalho, pois contém definições que são comuns em qualquer fábrica atual. Isto permite a criação de novas simulações muito rapidamente já que não são necessárias traduções entre a realidade e as aplicações simuladas. O treinamento dos projetistas no uso desta biblioteca também é bastante simplificado pelo mesmo motivo.

A biblioteca apresentada neste estudo é composta por diversas classes que implementam as várias entidades dos ambientes manufatureiros atuais e são interligadas de forma a assimilar o fluxo de trabalho nestas fábricas. Os ambientes reais possuem dois principais componentes: as entidades físicas e o controle destas entidades. As

entidades físicas, que correspondem a recursos existentes nas fábricas, são facilmente traduzidas para o paradigma orientado a objetos, já que praticamente cada classe corresponde a um tipo de recurso físico. Este é um dos motivos pelo qual a orientação a objetos está sendo tão tenazmente pesquisada para o uso em sistemas de manufatura. Por outro lado, o controle destas entidades não pode ser simplesmente colocado em uma classe, esperando que a mesma vá executar todas as funções de controle necessárias, pois a correspondência um para um não se apresenta neste caso. É neste ponto que se encontra a inteligência da biblioteca e onde reside a maior dificuldade de desenvolvimento, pois as interações entre as classes de controle e as classes de entidades físicas vão definir o que poderá e o que não ser modelado.

Esta biblioteca contém um conjunto de classes que permitem modelar fisicamente e logicamente um sistema de manufatura. A modelagem física é feita através de classes que representam os vários recursos de manufatura como máquinas, robôs e mão-de-obra. As classes apresentadas neste grupo não correspondem à totalidade dos recursos que podem ser encontrados nas fábricas atuais, mas o projetista pode, facilmente, e seguindo os padrões apresentados, especializar novas classes de recursos que irão refletir a aplicação requerida. A modelagem lógica consiste na utilização das classes e métodos propostos de modo a assimilar a lógica do modelo real, permitindo o controle das funções de manufatura.

Conforme vários estudos apresentados por outros autores quanto a diretivas usadas para a criação de hierarquias de classes, tentou-se reduzir ao máximo o acoplamento entre as classes, mas neste caso nem sempre foi possível devido à grande interação que existe entre as diversas entidades e seus controladores. Assim, as classes de controle lógico estão entre as classes de recursos físicos e não existe um delimitador visível que separe os dois tipos de classe. Inicialmente pode criar uma certa confusão ao projetista, mas como a abordagem de controle usada foi àquela baseada em roteiros e atividades, esta confusão é rapidamente desfeita.

A abordagem de controle apresentada neste trabalho, como dito anteriormente, é baseada em roteiros e atividades. Isto significa que o controle de produção é feito por roteiros de produção, que possuem atividades e vão acionar os respectivos recursos. Esta abordagem facilita o projeto, pois a atual teoria dos sistemas de manufatura é baseada neste tipo de proposta. Algumas outras classes de controle foram adicionadas para aumentar o poder de modelagem.

3.2 Especificação da Biblioteca Proposta

A biblioteca de classes proposta pode ser vista na figura 5, que representa o diagrama de classes. Esta biblioteca foi criada utilizando-se a linguagem UML, que é uma linguagem padrão para a modelagem de objetos (FOWLER, 2000 e BOOCH, 2000). Ela é composta por 30 classes interligadas por relacionamentos, agregações e especializações. A raiz desta hierarquia é a classe `Linha_Producao`. À esta classe são agregadas duas classes que executam funções distintas no modelo: `Recurso` e `Plano_Mestre`. A classe `Recurso` é a raiz a todas as classes de recursos físicos que podem executar um trabalho útil na peça, transformando-a de alguma forma ou conduzindo-a para outro local no sistema. A classe `Plano_Mestre` é a raiz das classes de controle da simulação. Nesta classe é gerado o plano de produção baseado nos pedidos dos clientes. Para criar o plano mestre, a classe `Plano_Mestre` necessita de informações de várias outras classes que a auxiliam neste processo. Estas classes são as

classes `Pedido`, `Algoritmo_Escalonamento`, `Peca` e `Recurso`, com sua agregação `Capacidade_Diaria`. O usuário interage com o modelo através da classe `Pedido`, inserindo pedidos de clientes que deverão ser entregues na data prevista e na quantidade requisitada pelo cliente. A classe `Algoritmo_Escalonamento` implementa os algoritmos de escalonamento de produção que irão guiar a criação do plano mestre. Estes algoritmos podem ser selecionados, adicionados e testados conforme a necessidade. Outra classe importante para a produção do plano mestre é a classe `Recurso`. Para escalonar a produção, o plano mestre usa informações sobre a capacidade dos diversos recursos existentes no sistema. Através da classe `Capacidade_Diaria` é possível saber o nível de utilização dos recursos envolvidos na produção de uma peça e calcular se é possível, com a capacidade atual, atender aos pedidos pendentes. A classe `Peca` deve fornecer informações sobre a peça a ser produzida, através das instâncias da classe atividade agregada a ela, a fim de possibilitar o cálculo da capacidade por recurso.

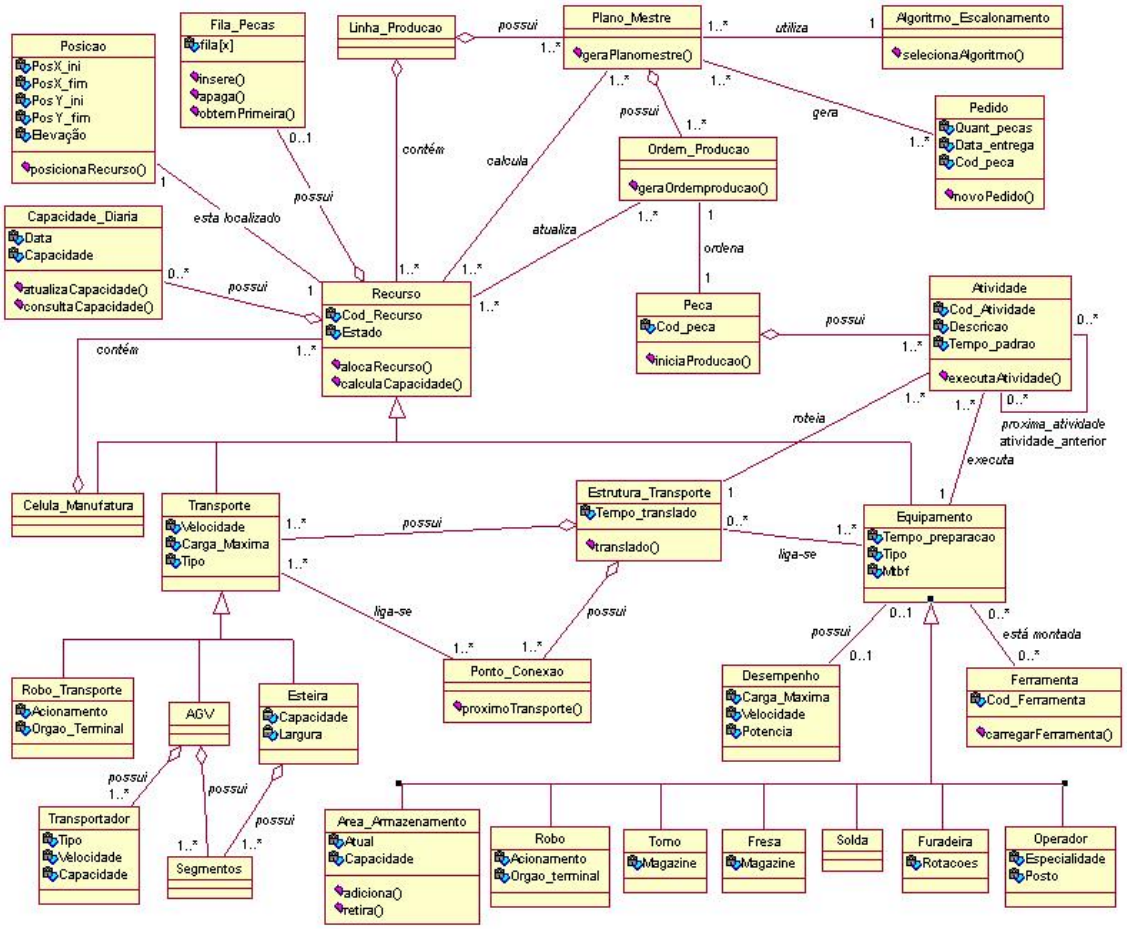

FIGURA 5 – Biblioteca de Classes proposta

A classe `Recurso` especializa-se em três classes que permitem a modelagem de entidades físicas que executam atividades e movimentam as peças de um local para outro. São elas a classe `Equipamento`, `Transporte` e uma classe especial chamada `Celula_Manufatura`, onde, juntamente com as classes `Recurso`, `Transporte` e `Equipamento`, implementam o padrão de projeto *Composite*. Este padrão permite compor objetos em estruturas de árvore para representarem hierarquias parte-todo e

permitir aos clientes tratarem de maneira uniforme objetos individuais e composição de objetos (GAMMA, 2000). Desta maneira pode-se ter uma classe `Celula_Manufatura` com uma construção mais complexa, onde estas células podem conter outras células ou outros recursos disponíveis. Esta abordagem pode ser usada até que o modelo corresponda à realidade.

A classe `Equipamento` divide-se em várias sub-classes que correspondem aos recursos especializados e específicos para cada operação. Como dito anteriormente, estas sub-classes não possuem o grupo completo de equipamentos que podemos encontrar em uma indústria de manufatura, porque seria impossível colocá-los todos em um diagrama. Na verdade, não é necessária a incidência dos recursos à exaustão, pois o projetista pode facilmente especializar novas entidades a medida que elas são necessárias. A classe `Equipamento` relaciona-se com uma classe chamada `Desempenho`. Esta classe tem a função de armazenar os atributos de desempenho dos diversos recursos a fim de ser possível a correta alocação da entidade ideal para realizar determinada atividade.

Por sua vez, a classe `Transporte` é responsável por modelar as entidades que têm a função de transportar as peças de um ponto a outro no chão de fábrica. Esta classe especializa-se em três outras: `Robo_Transporte`, `AGV` e `Esteira`. As esteiras e os AGV's possuem uma classe agregada chamada `Segmentos`. Esta classe permite a construção de vários segmentos não lineares de instâncias de esteiras e AGV's a fim de possibilitar a representação de sistemas de transporte complexos. Diferentemente das esteiras, que possuem uma construção similar a um trilho por onde as peças fluem, os AGV's possuem veículos de transporte que são guiados por trilhos ou sensores. Assim, foi necessária a inclusão de outra agregação à classe AGV, chamada de `Transportador`, que especifica as características do veículo de transporte usado naquele AGV em particular. As classes de transporte não possuem atividades pré-definidas visto que sua função é apenas de transportar de um local para outro, sendo desnecessária uma instância para especificar uma ação tão simples.

A classe `Ferramenta` permite que se atribua a um equipamento uma ferramenta para a realização de uma determinada atividade. Quando uma atividade é requisitada, a mesma deve controlar se a ferramenta necessária para a conclusão da operação está montada na máquina em questão. Isto é verificado através de um relacionamento entre as classes `Equipamento` e `Ferramenta`. Caso a ferramenta não esteja montada e posicionada corretamente no equipamento em questão, o atributo `Tempo_preparacao` da classe `Equipamento` deve ser contabilizado. Desta maneira a ferramenta adequada passa a estar montada no equipamento, liberando a atividade. A classe `Equipamento` também possui o atributo `Mtbf`, que representa o tempo médio entre falhas de um recurso. Com este atributo pode-se simular situações de falha nos recursos, tornando a simulação mais próxima da realidade.

A maioria das classes descritas até este ponto são responsáveis pela modelagem de entidades que representam recursos físicos nos sistemas de manufatura. Apesar destes recursos serem importantes para o modelo, os mesmos não têm funções de controle da produção. Como dito anteriormente, as classes que executam as funções de controle são a inteligência do modelo, onde se pode definir como a simulação irá se comportar.

O cerne do controle de produção na biblioteca apresentada é formado por um grupo de classes que juntas executam estas funções. São elas: `Pedido`, `Plano_Mestre`,

Algoritmo_Escalonamento, Peca, Atividade, Recurso, Capacidade_Diaria e Ordem_Producao.

Existem dois momentos bem definidos na operação da simulação: o primeiro momento é a entrada do pedido até a geração das ordens de produção. O diagrama de seqüência desta fase inicial é apresentado na figura 6. Em um segundo momento, estas ordens de produção disparam a produção das peças. O diagrama de seqüência da segunda fase é mostrado na figura 7.

3.2.1 Inserção dos Pedidos e Geração das Ordens de Produção

A classe Pedido e seu método novoPedido representam a interface de entrada da biblioteca. Instâncias desta classe armazenam os pedidos feitos pelos clientes e a partir deles será gerado o plano mestre que irá escalonar a produção de acordo com os atributos Quant_pecas, Data_entrega e Cod_Peca. Cada vez que um novo pedido é colocado no sistema, é gerado um novo plano mestre de acordo com as ordens de produção que já estão em execução e aquelas pendentes, além das informações de quantidade de peças e data de entrega das peças.

A classe Plano_Mestre é responsável por executar o escalonamento da produção usando informações dos pedidos pendentes, ordens de produção em execução, ordens de produção pendentes e as datas de entrega dos pedidos. O plano mestre pode ser executado utilizando-se qualquer um dos algoritmos de escalonamento de produção que estão em instâncias da classe Algoritmo_Escalonamento. O projetista pode criar várias instâncias de algoritmos e executar o escalonamento de produção com cada um deles com a finalidade de verificar qual deles apresenta a melhor solução.

O operador insere o pedido do cliente através do método novoPedido da classe Pedido, especificando os atributos Data_entrega, Quant_pecas e Cod_peca. Ao inserir o pedido, é disparado o método de reconstrução do plano mestre chamado geraPlanomestre na classe Plano_Mestre. Para isso, são necessárias várias informações provenientes de outras classes, que fornecem informações elementares para a construção do plano mestre. Inicialmente a classe Plano_Mestre deve selecionar um algoritmo de escalonamento de produção na classe Algoritmo_Escalonamento, usando o método selecionaAlgoritmo. De posse do algoritmo de escalonamento, o plano mestre deve consultar a capacidade dos recursos para determinar qual será a data de entrega do pedido do cliente. Este cálculo é feito com base nos pedidos já consolidados, os pedidos pendentes e as peças em produção. Com estas informações é possível definir a data aproximada de entrega e, se caso esta data for maior do que aquela contratada com o cliente, o sistema deve gerar uma resposta ao usuário indicando que, com a carga de produção atual e os recursos disponíveis, não é possível efetuar a entrega daquele pedido na data requisitada. Assim o operador pode tomar a decisão de manter a data, ou negociar com o cliente uma prorrogação na data de entrega.

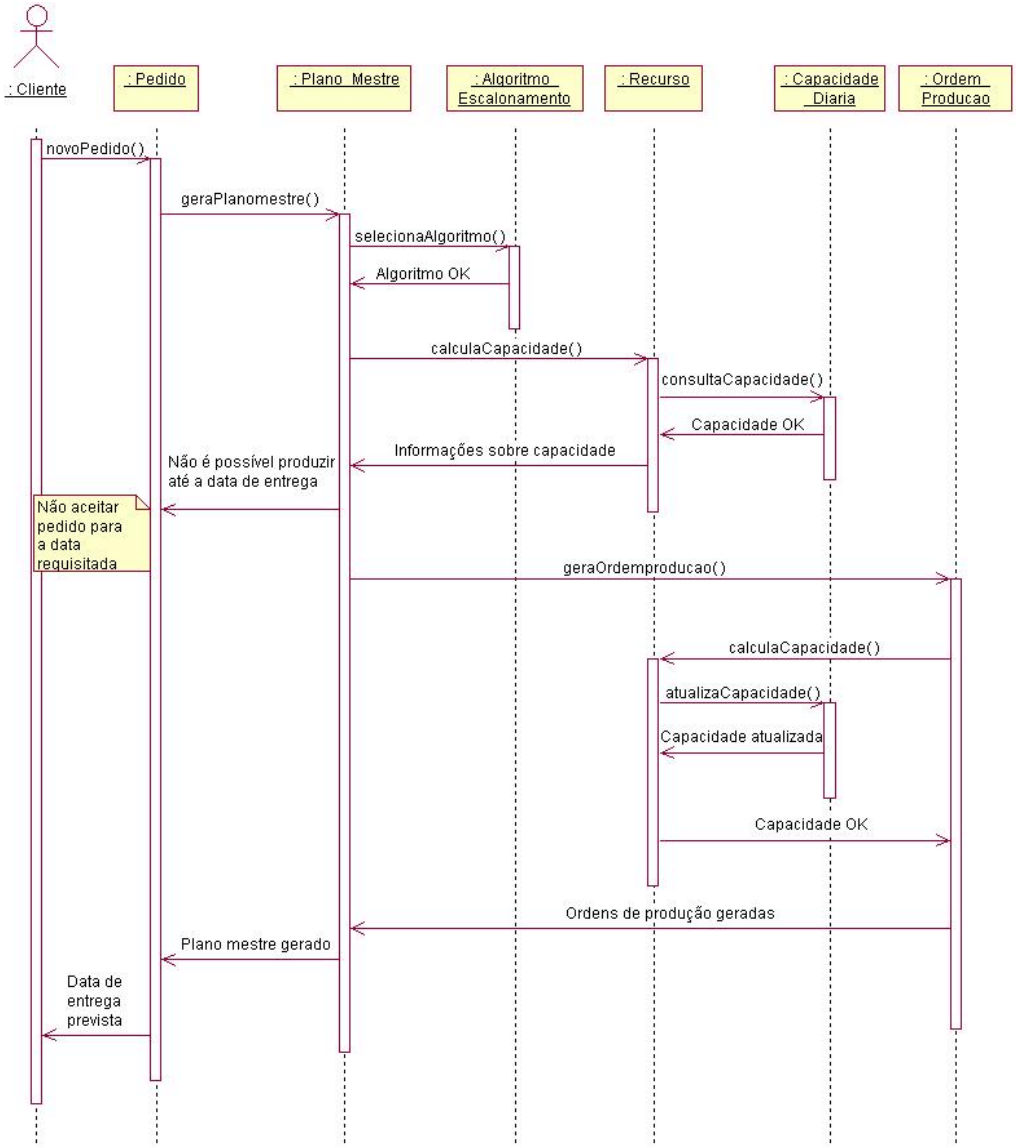

FIGURA 6 – Diagrama de Seqüência para a especificação das ordens de produção

Se, por outro lado, existe a possibilidade de produzir dentro do prazo estabelecido, o sistema automaticamente apresenta a possível data de entrega e gera as ordens de produção. As ordens de produção são geradas através do método `geraOrdemproducao`. Cada ordem de produção é responsável por ordenar a produção de uma instância da peça. Se, por exemplo, tivermos um pedido de 30 unidades de uma determinada peça, haverá 30 instâncias da classe ordem de produção geradas. Isto facilita o controle do fluxo de produção das peças. Além de gerar as ordens de produção, é necessário atualizar as informações de capacidade nos diversos recursos usados. Para isso, é feito recálculo de capacidade diária de cada recurso com base nos tempos de cada atividade que deverá ser executada, multiplicando estes valores pelo número de peças a ser produzida. Cada recurso possui uma capacidade de trabalho diária. Esta capacidade diária pode ser atualizada dependendo do ritmo de produção (usando horas extras, por exemplo). Quando esta capacidade é atingida, é criada outra instância da classe `Capacidade_Diaria`, indicando que a capacidade máxima daquele recurso para aquele dia foi alcançada. Quando as capacidades são atualizadas, as instâncias das ordens de produção são geradas, habilitando o início da produção.

3.2.2 Produção das Peças

A segunda fase da execução do modelo corresponde à simulação da produção das peças requisitadas. Na figura 7 pode-se visualizar o diagrama de seqüência desta fase que inicia com as instâncias das classes `Ordem_Producao` ordenando o inicio da produção de cada peça conforme o escalonamento feito pelo plano mestre. Através do método `iniciaProducao()`, da classe `Peca`, é disparada a produção das peças subordinadas a esta ordem. Cada ordem de produção está relacionada com uma instância correspondente da classe `Peca`, que, por sua vez, possui agregadas a ela, diversas atividades que irão controlar a operação dos respectivos recursos. Cabe ressaltar neste ponto a abordagem usada no controle da produção no modelo apresentado pela biblioteca. Como dito anteriormente, a inteligência da biblioteca está figurada no controle de todos os recursos físicos. Os recursos físicos podem ser mapeados um para um, diferentemente das operações de comando destes recursos. Esta biblioteca utiliza idéias provenientes dos próprios ambientes fabris. Nestes, o controle do seqüenciamento da produção de uma determinada peça é especificado em um documento chamado roteiro de produção. No roteiro de produção, estão listadas todas as atividades necessárias para completar a produção de um determinado produto. Também neste documento aparecem várias outras informações como as dependências de outras atividades, as ferramentas necessárias, os tempos padrão e de preparação das atividades, etc. Cada produto produzido possui um roteiro de produção associado a ele. Os roteiros de produção agregam muitos benefícios para o operador e para o projetista já que são uma importante forma de documentação do que deverá ser feito sobre o insumo para que o mesmo chegue no final da linha de produção com as formas e características requeridas.

Devido às suas características, uma variação do roteiro de produção foi utilizado nesta biblioteca para o controle da produção nos diversos recursos. O roteiro de produção é representado pela agregação da classe `Atividade` na classe `Peca`. As classes `Atividade` agregadas possuem um seqüenciamento lógico representado por um relacionamento indicando a próxima atividade e a atividade anterior. Uma atividade só pode ser executada se todas as atividades anteriores relacionadas a ela já foram executadas. Esta estrutura agregada forma um grafo de produção que pode controlar várias atividades em diversos níveis da hierarquia de produção. Por outro lado, quando uma tarefa é completada, a mesma chama a próxima tarefa cadastrada, ativando a próxima instância da classe Atividade. Se não houver mais atividades, é retornado um valor à classe peça indicando que a produção daquela peça foi concluída.

Cada instância da classe `Atividade` deve possuir um recurso padrão que irá controlar. Para isso, cada atividade possui um relacionamento com um recurso que irá executar aquela atividade. A instância da classe `Atividade` chama o método `aloca_recurso()` da classe `Recurso` passando para o método o tempo padrão de execução da tarefa. Associado a cada recurso foi adicionado uma fila de espera representada pela classe `Fila_Pecas`. Esta classe tem por função o enfileiramento de peças que foram escalonadas para produção pela classe atividade em um momento em que o recurso em questão já estava ocupado. A peça, ao encontrar o recurso ocupado, deve aguardar até que o mesmo esteja livre. Para isso, é usado o método `insere()` da classe `Fila_Pecas` que coloca um *token* em uma estrutura de dados do tipo fila dentro da classe `Fila_Pecas`. Quando o recurso passar do estado ocupado para livre com a finalização de uma atividade, se existirem *tokens* dentro da fila de peças, este deve

chamar o método `obtem_primeira()` que retira uma *token* que representa uma peça e repassa para a classe `Recurso`, mudando o estado do recurso de livre para ocupado.

A classe `Posicao`, relacionada à classe `Recurso`, indica o local físico onde encontra-se o referido recurso. Sua posição é especificada através dos atributos de posicionamento `posx_ini`, `posx_fim`, `pos_y_ini`, `posy_fim` e `elevação`. O uso desta classe não é obrigatório, pois a mesma não influencia a simulação. Esta classe torna-se interessante quando for necessária a especificação de posições físicas para a criação de diagramas de produção, leiautes de chão de fábrica e interfaces gráficas de simulação. Esta classe pode ser usada também para obter-se a distância linear entre dois recursos, podendo ser usada para cálculos de deslocamento e tamanho de recursos de transporte.

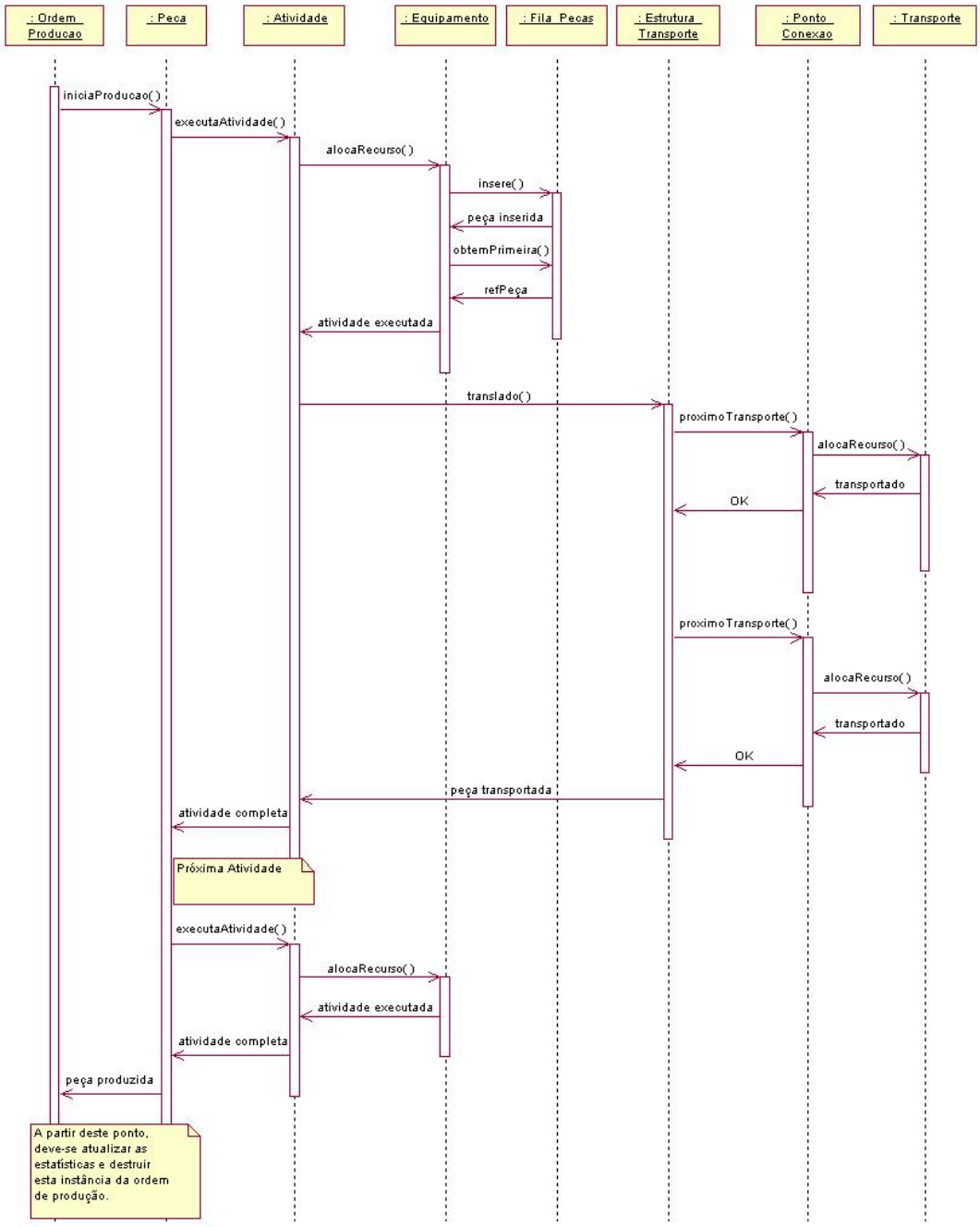

FIGURA 7 – Diagrama de Seqüência das interações de produção

3.2.3 Infra-estrutura de Transporte das Peças

O transporte das peças de um recurso para outro é uma função muito importante nos sistemas de manufatura, principalente quando se trata de um projeto flexível de manufatura. O transporte é uma ação necessária, mas não é especificada por uma atividade pois as funções de transporte são bastante simples, geralmente baseando-se no translado de uma peça de uma máquina para outra. Apesar desta ação ser simples, a definição dos recursos de transporte que farão o deslocamento e das rotas utilizadas pelas peças representam um dos mais intrincados problemas nos sistemas de manufatura flexíveis. Para resolver o problema do transporte de peças na biblioteca apresentada, utilizaram-se conceitos da disciplina de redes de computadores, que permite ao projetista especificar o nível de abstração que deseja alcançar em sua simulação.

Antes de apresentar a solução implementada na biblioteca, deve-se especificar o problema. Um sistema de manufatura necessita de recursos que efetuem transformações nas peças de modo que a mesma apresente as características físicas requeridas no projeto. Para isso, geralmente, várias atividades devem ser feitas sobre esta parte para que a mesma atinja sua forma final. O que se constata é que não existe uma máquina que consiga executar todas as atividades em um único local. Assim, é necessário transportar as peças em produção de uma máquina para outra a fim de completar a sequência de operações especificada no roteiro. Vários recursos podem realizar a tarefa de transportar uma peça de um local para outro. Dentre estes recursos pode-se citar as esteiras, os AGV's e até mesmo operadores humanos.

Em uma situação simples, como a mostrada na figura 8, não existe o problema de roteamento, pois a peça só tem um caminho a seguir: sai do recurso A, entra no transportador T (no caso uma esteira) e é levado até o recurso B.

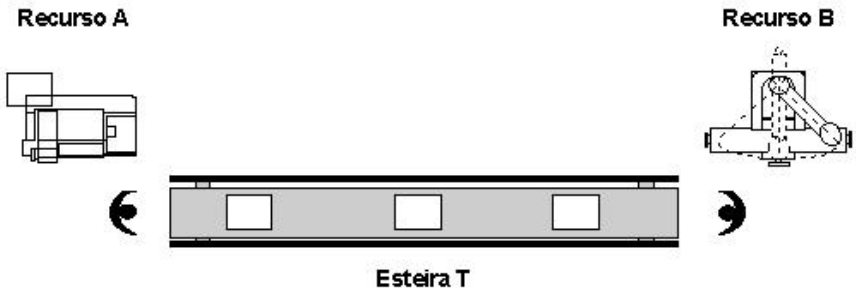

FIGURA 8 – Transporte entre dois recursos usando um único meio de transporte

Em uma situação hipotética, mais complexa, tem-se os mesmos recursos A e B de saída e de chegada, mas a peça para ir de um ponto a outro pode escolher vários caminhos e alguns destes caminhos possuem outros recursos intermediários, que podem ser outros recursos de transporte ou equipamentos de transformação. Pode-se visualizar esta situação na figura 9. A peça que sai do recurso A pode tomar 3 caminhos para chegar ao recurso B: via esteira, que passa por outro recurso intermediário e deve ser colocada em uma outra esteira de alguma forma, escolhendo o caminho correto; via AGV, com um transportador que leva a peça por um caminho mais comprido, evitando obstáculos; e via operador humano, que leva diretamente para o recurso B, uma de cada vez.

Baseando-se na segunda situação, pode-se concluir que o sistema de transporte pode ser tão complexo quanto necessário. Por outro lado, não se pode simplificar os esquemas de transporte sob pena de não ser possível a implementação de Sistema Flexíveis de Manufatura, que baseiam-se principalmente em seus recursos de deslocamento de materiais.

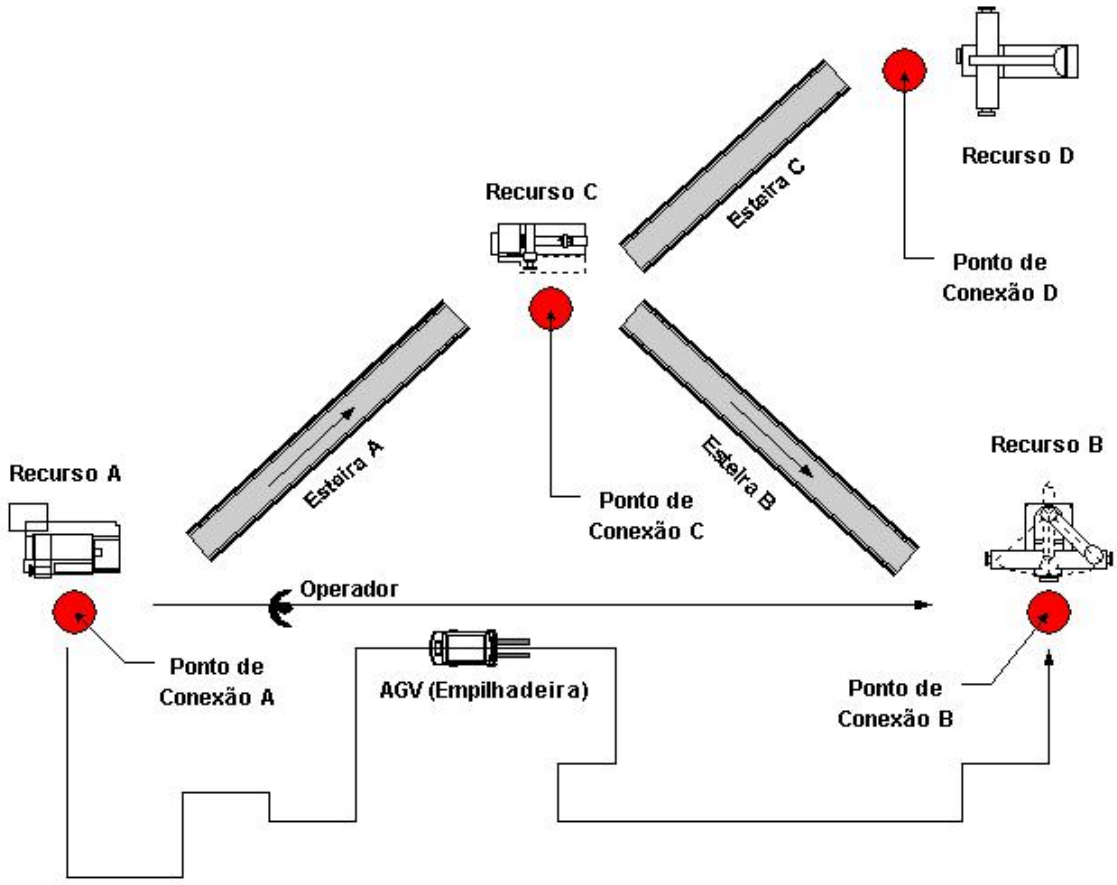

FIGURA 9 – Situação com caminhos alternativos para transferência de peças

Na biblioteca proposta, para ser possível a implementação de simulações com sistemas de transporte de materias complexos, foram utilizadas algumas classes que permitem, junto com os recursos físicos de transporte, controlar e rotear as peças para seus destinos corretos. Para representar os recursos físicos de transporte, são usadas classes especialisatas da classe `Transporte`, que são `Robo_Transporte`, `AGV` e `Esteira`. Outros recursos de transporte podem ser agregados ao modelo dependendo da necessidade do projetista. A representação do controle de rotas seguidas pelas peças é feita pelas classes `Ponto_Conexao` e `Estrutura_Transporte`.

A representação dos caminhos que uma peça pode seguir de um recurso de manufatura a outro utiliza a mesma concepção usada na Camada de Rede, da pilha de protocolos TCP/IP. Segundo Tanenbaum (1997), a camada de rede está relacionada à transferência de pacotes da origem para o destino. Para que se chegue ao destino, são necessários vários saltos por roteadores ao longo do percurso. Os roteadores são responsáveis por resolver o caminho que cada pacote deverá seguir. Cada pacote pode

seguir um caminho diferente do pacote anterior se existirem caminhos alternativos da origem ao destino.

Transportando o conceito das redes para os sistemas de manufatura, pode-se notar uma grande intersecção de funcionalidades: as peças no sistema representam os pacotes na rede. Os recursos de transporte como as esteiras e AGV's representam os caminhos físicos das redes como os cabos coaxiais e fibras óticas. Os pontos de conexão representam os roteadores nas redes. Os recursos de manufatura representam os hosts de origem e destino.

A classe `Estrutura_Transporte` permite encapsular todos os recursos físicos de transporte e todos os pontos de conexão. À esta classe são agregadas as classes `Transporte` e `Ponto_Conexao`. A estrutura de transporte permite que o projetista tenha uma visão mais abstrata do transporte, pois, para a classe `Atividade`, não importa quem realmente irá fazer o translado da peça de um recurso a outro. Contanto que os recursos físicos de manufatura sejam interligados por uma estrutura de transporte comum, o modo como a peça é conduzida fica transparente tanto para a classe `Atividade`, quanto para a classe `Equipamento`. As estruturas de transporte podem ser compostas por várias agregações de classes `Transporte` e `Ponto_Conexao`. Estas classes têm a função de conduzir e rotear as peças para os destinos corretos. Os pontos de conexão têm a função de resolver qual o melhor caminho que a peça pode seguir. Quando um recurso de manufatura termina sua atividade, este coloca a peça em um ponto de conexão. O ponto de conexão, por sua vez, vai escolher o próximo recurso de transporte a ser usado, podendo ser usado para esta escolha vários algoritmos de roteamento, como carga, velocidade, utilização, etc. Quando o ponto de conexão escolhe o próximo recurso que irá transportar a peça, a mesma é colocada no recurso que a desloca até o próximo ponto de conexão. Neste novo ponto, será feita uma nova escolha do melhor caminho a seguir. Quando a peça chegar no ponto de conexão correspondente ao recurso de manufatura correto, a máquina faz a carga da peça e começa a executar a próxima atividade. A figura 9 demonstra graficamente o exposto acima.

A biblioteca de classes apresentada neste capítulo foi descrita detalhadamente para permitir o correto entendimento do estudo de caso proposto nos próximos capítulos. O objetivo do estudo de caso é validar a biblioteca através de aplicações reais, comparando os dados obtidos a fim de avalizar os estudos realizados.

4 Estudo de Caso: Célula de Ponteiras da Empresa Pigozzi

Este capítulo tem por objetivo apresentar um estudo de caso que será usado para validar a biblioteca proposta, bem como sua versatilidade e características.

4.1 A Célula de Ponteiras

Este estudo de caso foi baseado na produção de Pontas de Eixo de Tratores, mostradas na figura 10, produzidas pela empresa Pigozzi S/A Engrenagens e Transmissões, localizada no município de Caxias do Sul, RS.

FIGURA 10 – Pontas de Eixo

A empresa Pigozzi S/A é fornecedora de conjuntos para montadoras agrícola, de transporte e mecânica pesada. Sua planta industrial está planejada de forma a assimilar a estratégia do negócio da empresa, tendo seu leiaute funcional arranjado de maneira tal que permita uma produção que reduza perdas e minimize custos (MARIANI, 2000). A produção de Pontas de Eixo está localizada dentro da linha de produção da empresa e é fabricada por uma célula de manufatura chamada de Célula de Ponteiras. A figura 11 representa um diagrama esquemático da Célula de Ponteiras.

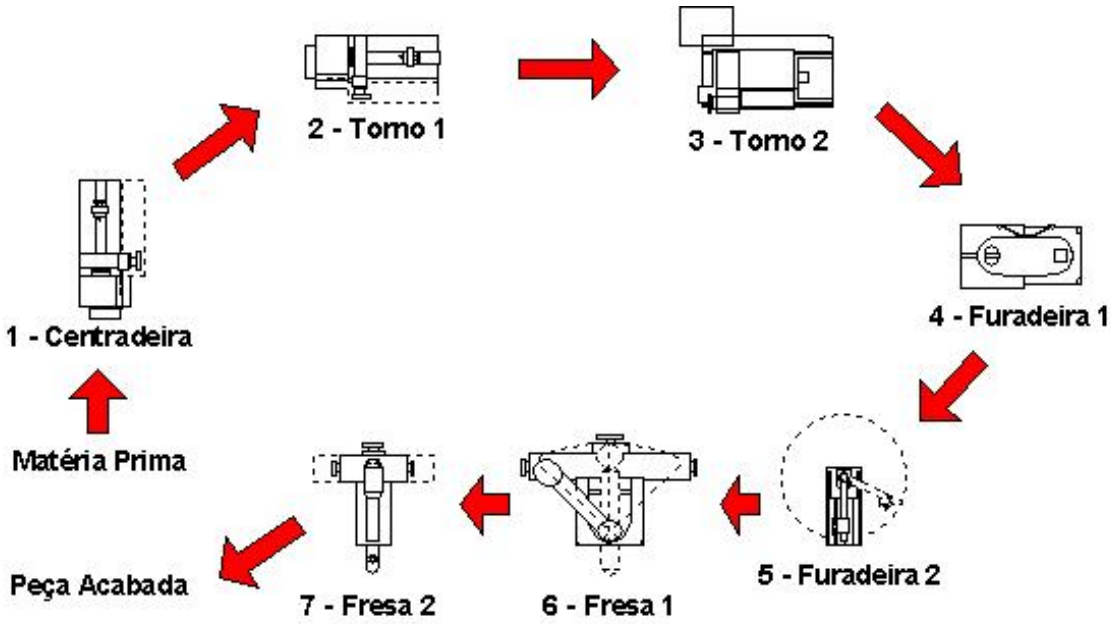

FIGURA 11 – Leiaute esquemático da Célula de Ponteiras

Apesar de ter uma estrutura fabril bastante sólida, esta empresa, na criação de seus leiautes de produção e células de manufatura, nunca utilizou técnicas como a simulação para obter os resultados que poderiam ser alcançados com a implantação de uma nova célula fabril. Além disso, os engenheiros não têm números precisos sobre os índices de ganho ou perda se algum recurso novo é adquirido ou mudado de lugar.

A simulação desta realidade, utilizando a biblioteca proposta, pode solucionar um problema que atinge grande parte do setor fabril: a determinação de gargalos e de pontos críticos para investimentos. Sem este auxílio, pode-se simplesmente tentar prever pontos problemáticos, sem, contudo, obter números precisos dos resultados.

A Célula de Ponteiras permite a realização de sete atividades distintas listadas na tabela 1:

TABELA 1 – Atividades da Célula de Ponteiras

Código	Máquina	Atividade	Tarefa
000225	Centradeira	Faceamento e Furo de Centro	A primeira tarefa desta máquina é o faceamento das extremidades da peça. Na segunda tarefa, é feito um furo em cada extremidade para permitir que o torno, na próxima operação, possa fixar a peça em seu berço.
000034	Torno Cop. Heycomat Aut.	Torneamento	Este torno realiza a retirada da carepa[1] deixada pelas operações de forjamento[2].
000474	Torno Horizontal CNC Daewoo Puma 350B	Torneamento	Esta operação realiza o torneamento de precisão na peça. Nesta operação, a peça toma sua forma bruta final, faltando apenas operações de acabamento.
000097	Furadeira Múltipla Brevet	Furação do Flange	Esta furadeira possui uma ferramenta especial de troca rápida. Cada ferramenta permite a furação de todos os furos do flange onde é fixada a roda do veículo. Esta ferramenta pode ser rapidamente trocada para permitir operações com vários tipos de pontas de eixo.
000148	Furadeira Radial Clever	Furação	Nesta operação são feitos furos de fixação do labirinto[3].
000083	Fresa Caracol ZEWZ	Edentamento	Cria os dentes da engrenagem[4] de tração da peça.
00068	Fresa Chaveteira WMW	Fresa do Rasgo	Cria na peça um rasgo na extremidade superior. Serve para fixar com a chaveta que está engatada na porca não deixando que a mesma solte-se ou afogue-se, evitando que a roda venha a cair, saindo do estriado.

Os dados para a aplicação do exemplo foram pesquisados em uma visita à linha de produção onde foram coletados textos, leiautes, roteiros e esquemas de produção da referida peça. As principais informações estão localizadas no roteiro de produção das

[1] Carepa é a descarbonização do metal após seu aquecimento.
[2] O Forjamento é o aquecimento do metal em 1200 a 1250 graus Celsius para conformização à quente dando a geometria desejada a fim de minimizar as operações nas peças.
[3] Labirinto é uma espécie de redutor que protege o conjunto da entrada de resíduos do trabalho.
[4] Os dentes da engrenagem são fixados nas estrias da coroa de redução final com o objetivo de ser arrastada junto com a roda que é fixada na ponteira para promover o deslocamento da máquina.

peças onde se encontra a lista de atividades que deverão ser executadas, bem como os tempos das atividades, tempos de preparação, ferramentas utilizadas e outras informações pertinentes à produção.

A célula de manufatura real foi projetada para permitir que mais de um tipo de ponta de eixo pudesse ser produzida utilizando-se as mesmas máquinas. Por isso, para concluir a fabricação de uma peça, a mesma não necessita passar por todas as máquinas, pois algumas peças podem necessitar ou não de alguma atividade realizada em uma determinada máquina. Por outro lado, existem operações que são feitas fora da célula de manufatura. Assim, é possível que uma peça tenha que sair da célula de ponteiras e ir para uma máquina ou processo externo. Um exemplo típico é o tratamento térmico e as operações de forja, que são impraticáveis de serem colocadas em uma mesma célula de manufatura devido à natureza de suas atividades.

Além das atividades realizadas na célula de ponteiras, outras operações são necessárias para a produção da peça em questão. Estas atividades não se encontram na célula estudada, mas devem pertencer ao modelo. Dentre elas tem-se: os desvios de processo para o tratamento térmico e operações de medição e calibração.

4.2 Modelagem da Célula de Ponteiras no Automod – Simulação do Roteiro 15228

A construção do modelo simulado do estudo de caso da célula de ponteiras da empresa Pigozzi começou na avaliação do material obtido junto à empresa. Os roteiros de produção, que especificam as atividades necessárias para a produção das peças, junto com o leiaute físico da célula foram utilizados para a modelagem do problema. A versão do modelo, baseado no roteiro de produção 15228 (referente a ponta de eixo 365), apresenta um processo serial e sem dependências de outras atividades, utilizando a maioria dos recursos da célula. O roteiro de produção referente a esta peça está apresentado no anexo 1. As especificações das tarefas de cada atividade do roteiro foram omitidas para proteger o processo produtivo da empresa. Esta simulação foi construída na versão 9.0 do Automod (AUTO SIMULATIONS, 2002).

Para representar os diversos recursos existentes no chão de fábrica, estes foram desenhados em escala utilizando-se suas medidas reais na ferramenta gráfica ACE (AUTO SIMULATIONS, 1999-c), que acompanha o simulador Automod. Esta ferramenta permite que o projetista desenhe, a partir de primitivas gráficas como cubos e cilindros, qualquer tipo de recurso, utilizando uma abordagem hierárquica. Na figura 12, pode-se visualizar a fresa Rm, a furadeira Brevet e um homem, que foram criados na ferramenta gráfica ACE e são utilizados no modelo simulado da célula de ponteiras.

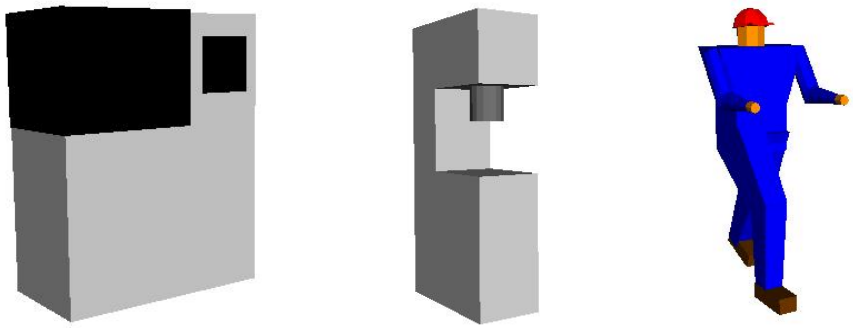

FIGURA 12 – Figuras representativas dos recursos de simulação

O Automod é um simulador orientado a processos, onde seu método de simulação baseia-se no translado das cargas, representando o trabalho em processo, pelos diversos recursos e entidades de transporte (AUTO SIMULATIONS, 1999-a/b). Para isso, o projetista deve controlar o tráfego das cargas no sistema através da especificação dos processos, que indicam ao mesmo quais o caminho que a carga deve seguir. Esta abordagem dá ao simulador características de robustez e confiabilidade, permitindo a construção de modelos bastante complexos. O problema é que, para isso, são necessários projetistas especializados, que dominem vários assuntos, como a teoria da simulação discreta, algoritmos e programação, noções de desenho técnico e a teoria dos sistemas de manufatura. Isto faz com que as empresas adotem geralmente métodos de simulação empíricos que dificilmente trazem os resultados esperados, resultado do superdimensionamento ou do sub-dimensionamento dos recursos, o que causa gastos duplicados ou desnecessários na aquisição de máquinas e recursos humanos.

Para criar um modelo no Automod são necessários vários passos. É recomendado, antes de iniciar a construção do modelo, executar uma análise detalhada sobre a realidade que se está querendo simular, a fim de definir o escopo, suas entidades e processos. Feito isso, é necessário criar o ambiente estático onde o modelo será criado. O ambiente estático consiste em entidades que não se movem e não executam nenhuma atividade no modelo. A sua única funcionalidade é definir limites físicos ou representar objetos que existem no modelo, mas não possuem especificação funcional. Para especificar um ambiente estático, devemos criar um novo sistema (New System) e especificá-lo como estático (Static). No estudo de caso da célula de ponteiras, foi especificado um ambiente estático para representar o chão do modelo.

No segundo passo, é necessário posicionar os recursos em seu local correto. Os recursos, neste caso, referem-se a máquinas e equipamentos que executam alguma modificação física na peça (ou cargas, como o Automod prefere referenciar). Para fazer isso, é necessário especificar um novo sistema chamado pelo Automod de *Process System*. Este sistema é o mais importante na simulação, já que é nele onde são especificados os recursos, filas, processos, procedimentos, etc. Na figura 13 é mostrado o menu do sistema Processo. Utilizando a opção *Resources*, pode-se especificar os recursos do modelo. No caso da célula de ponteiras, foram especificados nove recursos representando cada máquina da célula, que são mostrados na figura 14. Para posicionar os recursos criados em seu local no modelo, importa-se a representação gráfica previamente construída em uma ferramenta gráfica como a que acompanha o Automod (ACE) e especifica-se seus atributos de posição no modelo. No estudo de caso das células de ponteira, cada recurso foi posicionado em cima do chão (entidade estática),

de acordo com as especificações contidas na planta baixa do chão de fábrica, respeitando as escalas e distâncias de modo que o modelo representasse com perfeição a realidade. Na figura 15, são mostrados os vários recursos posicionados em seus devidos lugares de acordo com os diagramas.

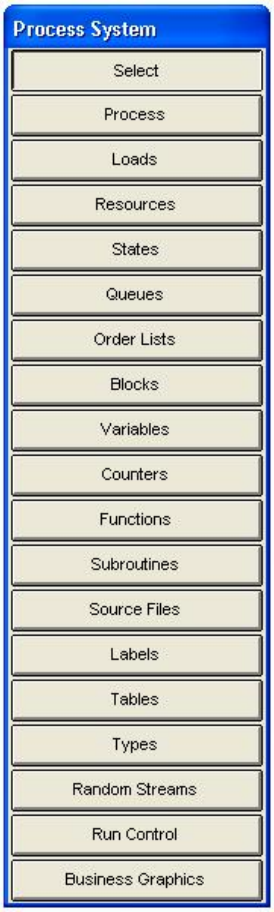

FIGURA 13 – Menu *Process System*

O próximo passo é a especificação das entidades que irão transportar as peças de uma máquina para outra. O Automod permite a simulação de diversas estruturas de transporte complexas como AGV's, esteiras, pontes rolantes, AS/RS, etc. O projetista deve incluir um novo sistema no modelo de acordo com a funcionalidade de transporte desejada. Na célula de ponteiras são usados apenas seres humanos que são responsáveis por carregar e ajustar as peças nas máquinas e transportar as peças de uma máquina para outra. São necessários 4 homens para executar estas tarefas. Como cada homem pode executar trajetórias diferentes e independentes, foi necessária a criação de quatro sistemas do tipo AGV. Foram usados os AGV's para especificar as trajetórias humanas porque tanto um veículo como um homem possui características, neste caso, similares.

FIGURA 14 – Janela *Resources*

Após a criação dos 4 sistemas de transporte, nomeados como `Worker_1`, `Worker_2`, `Worker_3` e `Worker_4`, em cada um foi criada a trajetória que cada homem deve realizar. Além disso, o Automod prevê a utilização de Pontos de Interação, que fazem a ligação entre os recursos e os pontos de entrada nos sistemas de transporte. Deve-se também importar a representação gráfica do homem que irá ser mostrada no modelo. A visualização gráfica do homem pode ser vista na figura 12. Com isso, a construção do modelo físico está completa, faltando a parte mais importante, que é a especificação funcional do modelo. Esta especificação é feita através de uma linguagem de programação proprietária, com sintaxe própria.

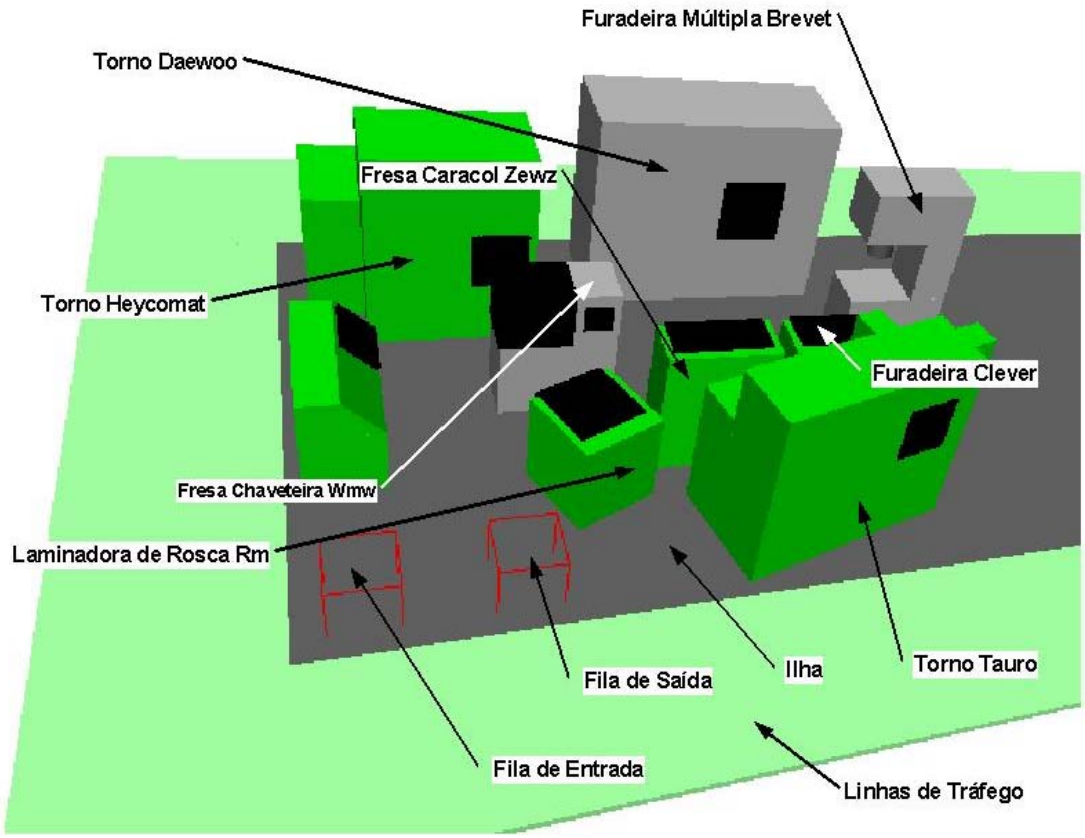

FIGURA 15 – Disposição dos recursos na área de simulação

O Automod possui uma abordagem orientada a processos. Por meio deles, o projetista é capaz de especificar o controle da simulação indicando ao simulador qual é a seqüência lógica de passos que uma peça (carga) deve descrever. A peça, por sua vez, entra no sistema e é enviada de processo em processo, executando as tarefas especificadas na linguagem de programação proprietária. Pode-se comparar um processo com um procedimento. Quando a peça é enviada a um determinado processo, este a processa de acordo com o algoritmo contido no procedimento. No último processo, a peça sai do sistema, atualizando suas estatísticas.

Para realizar a especificação funcional do modelo, o primeiro passo é criar os vários processos que irão fazer o tratamento das cargas que utilizam os recursos. Utilizando-se o menu *Process System*, opção *Process*, pode-se criar os processos necessários. Cada processo possui uma função especial denominada *Arriving Procedure*, que é um procedimento que é executado cada vez que uma peça entra no escopo do processo em questão. Neste procedimento é feita a especificação, através da linguagem de programação proprietária, de quais ações a peça deverá realizar enquanto estiver neste processo, até ser enviada a outro processo, ou sair do sistema. Na figura 16, é apresentado trecho do código usado na modelagem da célula de ponteiras.

```
1    begin P_Brevet arriving procedure
2      move into Q_Brevet
3      use Brevet for normal 480,48 sec
4      move into Worker_2:cp_brevet
5      travel to Worker_2:cp_clever
6      send to P_Clever
7    end
```

FIGURA 16 – Trecho de código do procedimento de chegada do processo `P_Brevet`

A primeira e a última linha apresentam o cabeçalho e delimitadores do procedimento. Na linha 2, a peça é conduzida até a fila de entrada do recurso. Se o recurso estiver ocupado, a peça aguardará nesta fila até que o recurso esteja livre. Após, na linha 3, o procedimento ordena o processamento da peça usando o recurso e tempo especificados. Pode-se usar várias distribuições estatísticas geradas pelo programa. No caso, foi usada uma distribuição Normal, com 480 segundos e variação de 10% para mais e para menos. Neste tempo, o recurso fica bloqueado efetuando o processamento da peça. Logo após, o processo move a peça até o ponto de interação entre o recurso e o sistema de transporte (no caso um AGV), ativando o veículo (no caso um homem), a buscar a peça e transportá-la até o próximo ponto de interação (linha 5). Chegando no próximo ponto de interação, a peça é enviada para o próximo processo (linha 6) ou é excluída do sistema (se for o último processo).

As cargas são especificadas através do menu *Process System*, opção *Loads*. Nesta opção o Automod permite criar novas cargas para serem conduzidas no sistema. Para isso, deve-se especificar várias opções como o processo inicial, o tempo de criação entre cada carga e a forma física da peça, que pode ser importada de um desenho da ferramenta gráfica ACE.

Para executar o modelo, o mesmo deve ser compilado através da opção *Run Model* no menu *Model*. Após a compilação, o Automod apresenta o modelo construído e pronto para a execução. Ao ser iniciada a simulação, as peças vão entrando no sistema

e atravessando os processos, executando os procedimentos de cada um. Durante a simulação pode-se a qualquer momento visualizar as estatísticas produzidas pelo programa que vão desde a utilização dos recursos até a criação de gráficos para análise dos resultados. Na figura 17, é mostrado o modelo em simulação. A ferramenta de simulação é dividida em três janelas. Na primeira, e maior, aparece o modelo gráfico, onde é possível visualizar o andamento da simulação. Esta tela apresenta menus onde se pode selecionar vários relatórios em tempo real do curso da simulação. Na parte inferior desta tela, o programa mostra o tempo de simulação atual. Existem outras duas janelas auxiliares. Uma delas fornece mensagens de controle da simulação e a outra o estado atual da mesma.

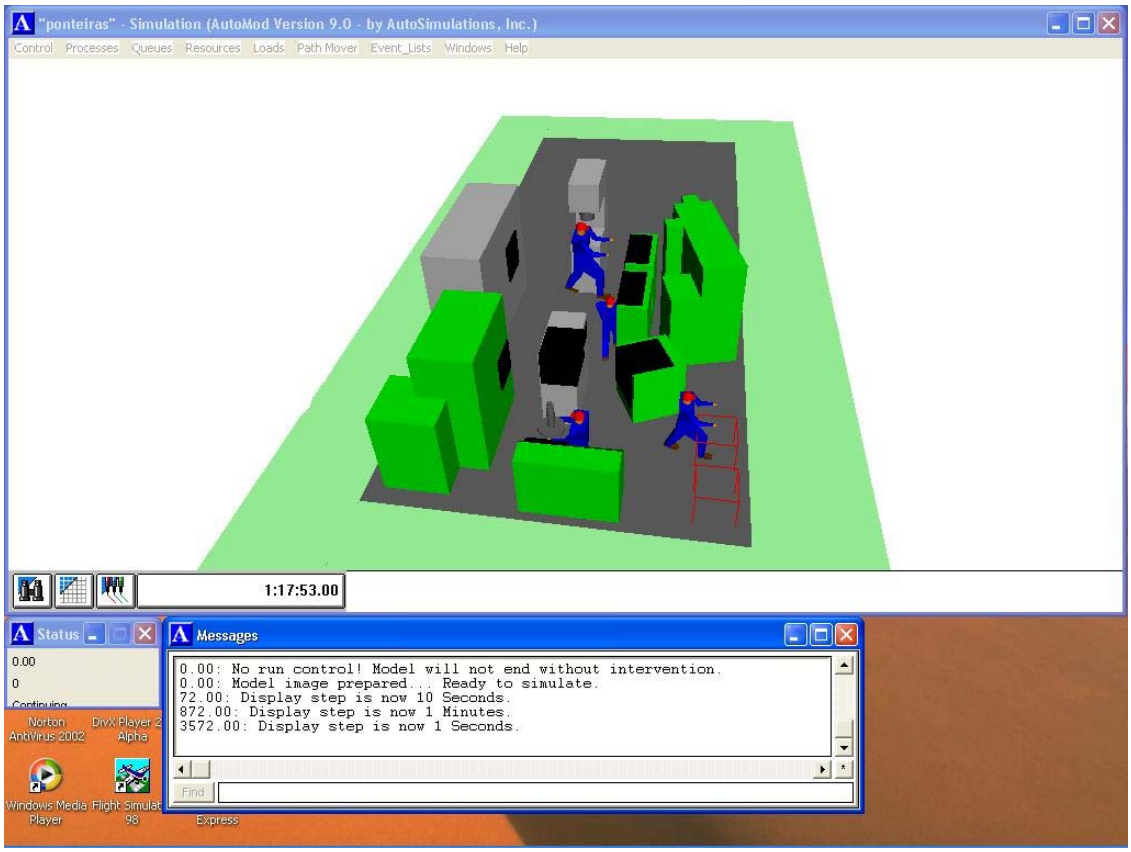

FIGURA 17 – Modelo em simulação

Analisando os resultados obtidos com a simulação deste estudo de caso no Automod pôde-se verificar que os mesmos são muito parecidos com a produção real das ponteiras na célula de manufatura. Os tempos totais de produção (lead-time), tempos de espera, porcentagens de utilização das máquinas e tempo total das peças na fila de entrada dos recursos foram mensurados e em nenhum dos resultados colhidos no Automod diferiram em mais de 10% para mais ou para menos do que se verifica na célula real. O Automod pôde também apontar com perfeição o recurso gargalo, função esta muito importante para a otimização dos sistemas de manufatura. Com isso conclui-se que o Automod é uma ferramenta bastante poderosa para a finalidade proposta, desde que diversas premissas sejam perfeitamente cumpridas. Dentre estas premissas podemos citar o nível de especialização do projetista, uma correta análise dos dados de entrada e um modelo perfeitamente construído que espelhe corretamente o sistema real.

4.3 Modelagem da Célula de Ponteiras usando a Biblioteca de Classes Proposta - Modelagem do Roteiro 15228

A biblioteca de classes descrita no capítulo 3 representa a base da metodologia orientada a objetos para modelagem de sistemas de manufatura apresentada neste trabalho. Nesta seção será criada a modelagem do roteiro de produção 15228 (referente a ponta de eixo 365), da célula de ponteiras da empresa Pigozzi, usando a biblioteca de classes proposta.

O estudo do leiaute e da construção da célula de ponteiras e dos roteiros de produção fornece insumos suficientes para criar o modelo em questão. O diagrama de instâncias completo está exposto no anexo 2. Os diversos recursos físicos como máquinas, ferramentas, transportadores e a mão de obra humana são representados por instâncias das sub-classes da classe `Recurso`. O controle do modelo é feito pelo roteiro de produção que contém as diversas atividades necessárias para a produção da peça.

O diagrama de instâncias é constituído pelos diversos recursos que irão executar atividades de transformação na peça. Estes recursos são as diversas máquinas como Fresas, Tornos e Furadeiras. Como cada diagrama de instâncias representa a modelagem da produção de uma peça, não é necessária a instanciação de máquinas que pertencem à célula de manufatura, mas que não serão usadas na produção da peça em questão. Neste caso, foram criadas 10 instâncias representando cada recurso. Além dos recursos deve-se adicionar ao diagrama as instâncias das atividades que devem coordenar a execução nos recursos e também as instâncias das classes de transporte que são responsáveis pelo translado de materiais de um ponto a outro do sistema.

Apesar dos diagramas apresentados (classes, instâncias e seqüência) serem excelentes ferramentas de modelagem, eles não permitem que se faça uma validação concreta da biblioteca. Para permitir que todas as idéias apresentadas na forma de diagramas sejam realmente aceitas, é necessário verificar a validade da mesma não só no âmbito conceitual através dos diagramas. É necessário também que sejam implementadas as idéias aqui contidas visando a possibilidade de diversos tipos de comparações, para só então ser possível sua aceitação.

A validação do modelo conceitual da biblioteca apresentada no capítulo 3, e do estudo de caso da empresa Pigozzi, simulado no Automod, foi baseada na implementação de um código na linguagem de programação C++ (DEITEL, 2001). Para a geração do código, foi usada a ferramenta *Case* Rational Rose 98 (RATIONAL SOFTWARE CORPORATION, 1998). Depois de gerados os códigos, foram feitas algumas modificações no mesmo visando uma melhor inteligibilidade. Também foram codificados os corpos das funções membro utilizando-se para isso as interações apresentadas nos diagramas de seqüência. Algumas das classes do modelo não foram codificadas porque não se mostram necessárias para a implementação do estudo de caso apresentado. Outras classes, como as classes `Relogio`, `Entidade_Simulada` e `NodoListaAtividade` foram adicionadas na implementação como o objetivo de controlar o tempo da simulação e criar uma fila de atividades agregada a classe `Peca`. O código completo das classes é apresentado no anexo 3. Para gerar os arquivos executáveis foi utilizado o ambiente de programação da Microsoft Visual C++ em sua versão 6.0 (MICROSOFT CORPORATION, 1998). A seguir são apresentadas algumas características das classes implementadas na simulação.

↳ Classe `Area_Armazenamento`

Classe derivada da classe `Equipamento`. Representa as áreas de armazenamento de materiais. Pode representar kanbans, AS/RS, almoxarifados, etc.

Atributos: `Capacidade` (inteiro) e `Atual` (inteiro).

Funções Membro	Função
`int getCapacidade()`	Obter a capacidade total da área.
`int GetAtual()`	Obter a capacidade atual da área.
`bool Adiciona()`	Colocar uma peça na área.
`bool retira()`	Retirar uma peça na área.

↳ Classe `Atividade`

A classe `Atividade` tem a função de armazenar informações sobre as atividades que devem ser executadas nas peças. É uma agregação da classe `Peca`.

Atributos: `Cod_Atividade` (inteiro), `Descricao` (caractere), `Tempo_Padrao` (inteiro), `refEquipamento` (ponteiro para `Equipamento`) e `refTransporte` (ponteiro para `Estrutura_Transporte`).

Funções Membro	Função
`bool executaAtividade(bool)`	Executar a atividade especificada nesta classe.
`Equipamento* criaRelacionamentoEquip()`	Criar o relacionamento entre a classe `Atividade` e `Equipamento`.
`Estrutura_Transporte* criaRelacionamentoTransp()`	Criar o relacionamento entre a classe `Atividade` e `Estrutura_Transporte`.
`Int getCod_Atividade()`	Obter o código da atividade.
`char* getDescricao()`	Obter a descrição da atividade.
`int getTempo_Padrão()`	Obter o tempo padrão da atividade.

↳ Classe `Equipamento`

Classe derivada da classe `Recurso`. A classe `Equipamento` tem a função de representar os diversos recursos de manufatura.

Atributos: `Tempo_preparacao` (inteiro) e `Tipo` (caractere).

Funções Membro	Função
`int getTempo_preparacao()`	Obter o tempo de preparação do equipamento.
`int GetTipo()`	Obter o tipo do equipamento.

↳ Classe `Estrutura_Transporte`

Esta classe encapsula os recursos de transporte e os pontos de conexão que executam o translado das peças. Através dela, pode-se transportar uma peça de um ponto a outro sem precisar especificar ou transparecer os recursos de transporte que realmente o executaram.

Atributos: `Tempo_translado` (inteiro).

Funções Membro	Função
`int getTempotranslado()`	Obter o tempo de translado da peça de um local a outro.
`bool Translado()`	Efetuar o transporte de um recurso a outro usando tempo especificado no atributo `Tempo_translado`. Foi implementado nesta função membro uma distribuição normal que permite randomizar tempos de translado de mais ou menos 10% do tempo total.

↳ Classe `Fresa`

Classe derivada da classe `Equipamento`. Representação de uma máquina do tipo fresadora.

Atributos: `Magazine` (inteiro).

Funções Membro	Função
`int getMagazine()`	Obter o número de ferramentas carregadas no magazine da máquina.

↳ Classe `Furadeira`

Classe derivada da classe `Equipamento`. Representação de uma máquina do tipo furadeira.

Atributos: `Rotacoes` (inteiro).

Funções Membro	Função
`int getRotacoes()`	Obter o valor de rotações por minuto impressa pela broca instalada na furadeira.

↳ Classe `Laminadora`

Classe derivada da classe `Equipamento`. Representação de uma máquina do tipo laminadora.

Atributos: `Dentes` (inteiro).

Funções Membro	Função
`int getDentes()`	Obter o número de dentes da ferramenta que produz os dentes das engrenagens.

↳ Classe NodoListaAtividade

Classe auxiliar que implementa a fila de atividades. Esta fila é apontada por um ponteiro na classe `Peca`. Simula um roteiro de produção.

Atributos: `RefAtividade` (ponteiro para `Atividade`), `ProximaAtividade` (ponteiro para `NodoListaAtividade`), `Header` (ponteiro para `NodoListaAtividade`), `Fim` (ponteiro para `NodoListaAtividade`) e `Atual` (ponteiro para `NodoListaAtividade`).

Funções Membro	Função
`bool insereAtividade(Atividade*)`	Inserir uma atividade na lista de atividades.
`Atividade* proximaAtividade(bool)`	Selecionar a próxima atividade.
`bool ultimaAtividade()`	Retornar verdadeiro se for a última atividade da lista.
`void voltaInicioLista()`	Retornar ao início da lista.

↳ Classe Peca

Representa a peça que está sendo manufaturada. Especifica um ponteiro para a lista que contém as atividades.

Atributos: `Cod_peca` (inteiro) e `refRoteiro` (ponteiro para `NodoListaAtividade`).

Funções Membro	Função
`int getCodpeca()`	Obter o código da peça.
`bool iniciaProducao(int)`	Controlar a execução da produção. Contém um laço que conta o número de peças produzidas e outro laço que serializa as atividades.
`void criaRoteiro(NodoListaAtividade*)`	Criar o relacionamento entre a classe `Peca` e `Atividade`. Simula a criação de um roteiro de produção.

↳ Classe Pedido

É a interface da biblioteca com o usuário. Através de uma função membro o usuário insere o pedido do cliente, disparando a produção das peças.

Atributos: `Data_dia` (inteiro), `Data_mes` (inteiro), `Data_ano` (inteiro), `Quant_pecas` (inteiro), `Cod_peca` (inteiro) e `refPeca` (ponteiro para `Peca`).

Funções Membro	Função
`Tdata getDataEntrega()`	Obter a data de entrega do pedido.
`int getQuantpecas()`	Obter a quantidade de peças do pedido.
`int getCodpeca()`	Obter o código da peça do pedido.
`bool novoPedido(int,int,int,int,int)`	Inserir novo pedido.
`void criaRelacionamento(Peca*)`	Criar o relacionamento entre a classe `Pedido` e `Peca`.

↳ Classe `Recurso`

Classe pai das classes `Transporte` e `Equipamento`.

Atributos: `Cod_recurso` (inteiro) e `Estado` (inteiro).

Funções Membro	Função
`int getCod_recurso()`	Obter o código do recurso.
`int getEstado()`	Obter o estado do recurso. 0=LIVRE e 1=OCUPADO.
`bool aloca_recurso(int)`	Alocar o recurso utilizando-o pelo tempo determinado no atributo `Tempo_padrao` da classe `Atividade`. Foi implementado nesta função membro uma distribuição normal que permite randomizar tempos de translado de mais ou menos 10% do tempo total.

↳ Classe `Torno`

Classe derivada da classe `Equipamento`. Representação de uma máquina do tipo torno.

Atributos: `Magazine` (inteiro).

Funções Membro	Função
`int getMagazine()`	Obter o número de ferramentas carregadas no magazine da máquina.

↳ Classe `Transporte`

Classe derivada da classe `Recurso`. A classe `Transporte` tem a função de representar os diversos recursos de transporte.

Atributos: `Velocidade` (inteiro), `Carga_maxima` (inteiro) e `Tipo` (caractere).

Funções Membro	Função
`int getVelocidade()`	Obter velocidade de transporte do recurso.
`int getCarga()`	Obter carga máxima do recurso.
`char* getTipo()`	Obter o tipo do recurso.

↳ Classe `Relogio`

Classe que armazena o tempo da simulação.

Atributos: `Tempo` (inteiro).

Funções Membro	Função
`int getTempo()`	Obter o tempo do relógio.
`void incrementaRelogio(int)`	Incrementar o tempo conforme operações.

↳ Classe `Entidade_Simulada`

Classe pai de todas as classes que deverão utilizar serviços da classe `Relogio` para controlar o tempo da simulação. Possui um relacionamento com a classe `Relogio`.

Atributos: `refRelogio` (ponteiro para `Relogio`).

Funções Membro	Função
`void criaRelacionamento(Relogio*)`	Criar o relacionamento com a classe `Relogio`.
`void simula(int)`	Chamar a função membro da classe `Relogio` que incrementa o tempo de simulação.

Para simular o estudo de caso usando a implementação da biblioteca na linguagem C++, são necessários quatro passos bem definidos que são implementados na função `main`. O primeiro passo é instanciar todas as classes necessárias na simulação conforme o diagrama de instâncias, passando a seus respectivos construtores as informações pertinentes aos recursos modelados. O modo de instanciação das peças de das atividades é mostrada na figura 18. A figura 19 mostra a instanciação dos recursos físicos de manufatura e transporte.

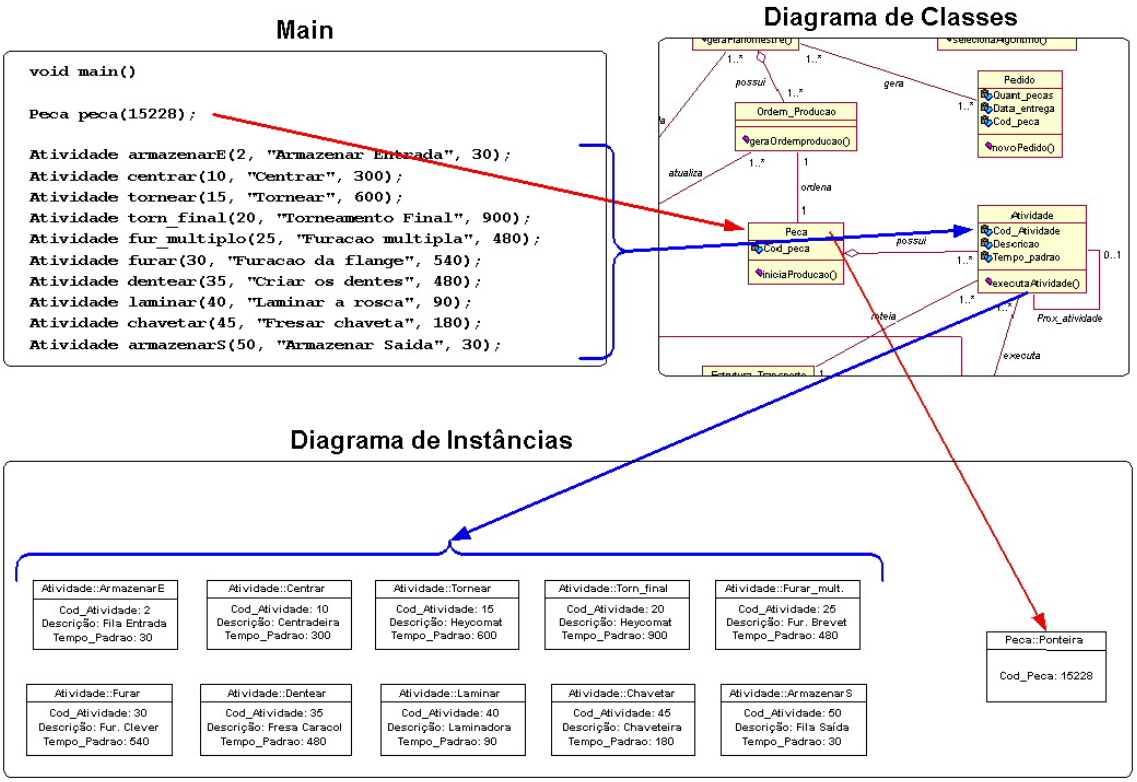

FIGURA 18 – Relação entre o *Main*, o Diagrama de Classes e o Diagrama de Instâncias na instanciação das Atividades e Peças

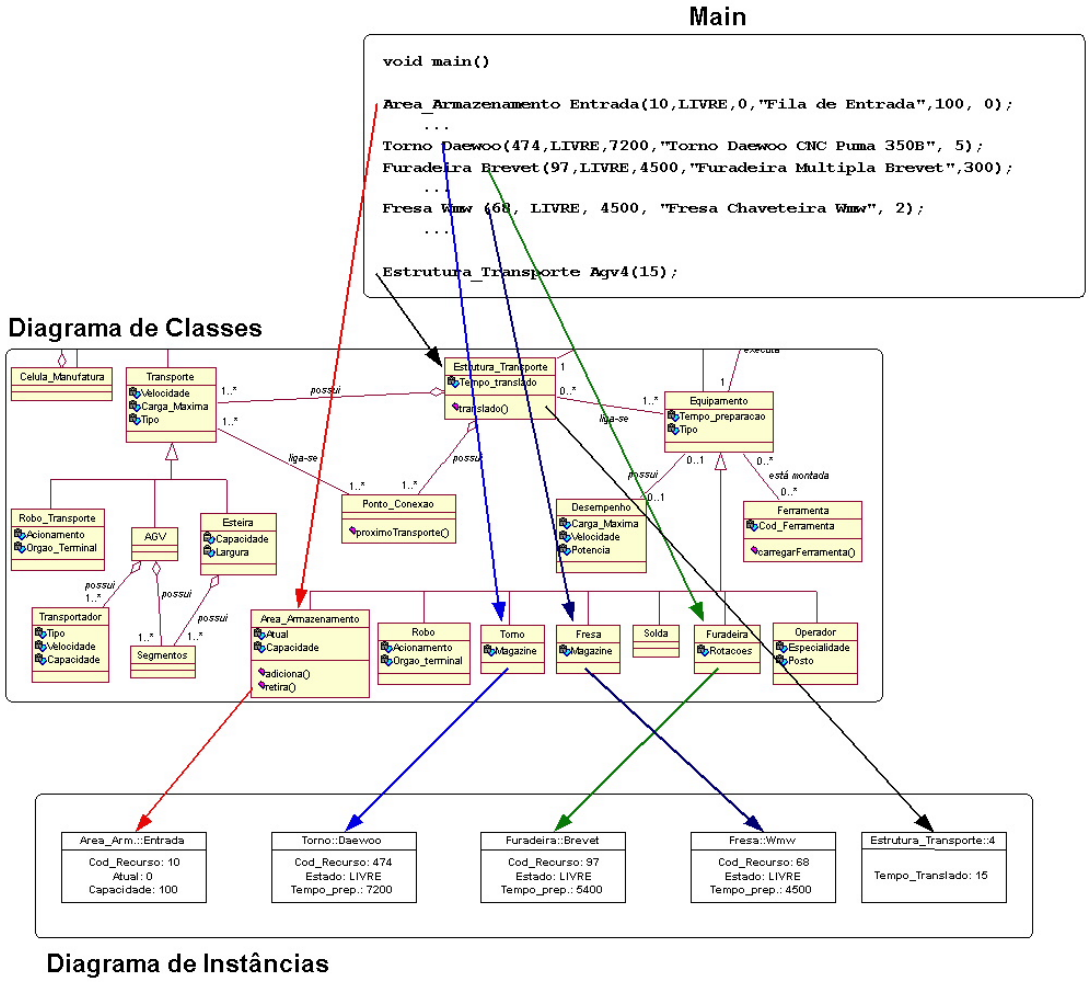

FIGURA 19 – Relação entre o *Main*, o Diagrama de Classes e o Diagrama de Instâncias na instanciação dos recursos de manufatura e transporte

A segunda fase é a de criação e encadeamento das atividades no roteiro de produção. Esta fase é muito importante, pois é nela que está a seqüência de operações correta que vai efetuar a produção da peça. Esta fase é mostrada na figura 20.

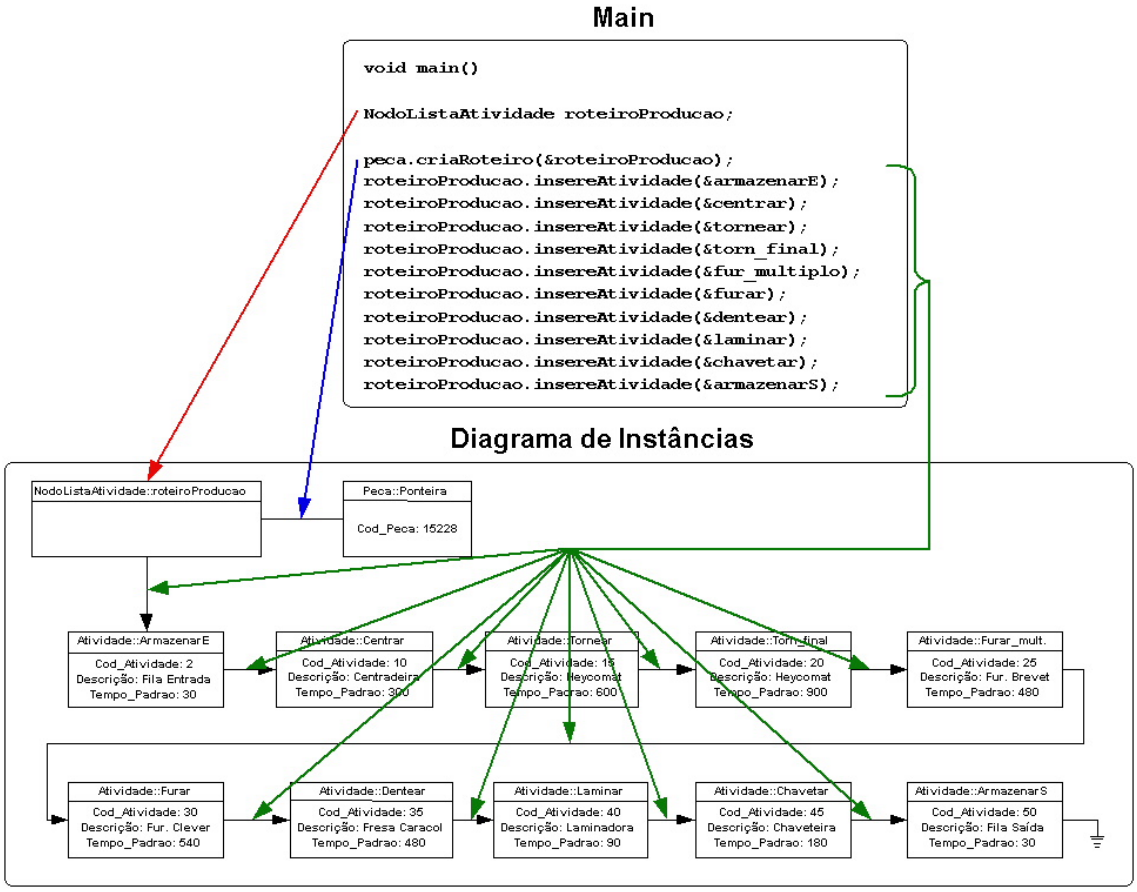

FIGURA 20 – Relação entre o *Main* e o Diagrama de Instâncias na criação do Roteiro de Produção

A terceira fase consiste na criação dos relacionamentos entre as diversas instâncias da simulação. A parte importante desta fase é especificar os relacionamentos corretos das atividades com seus respectivos equipamentos. A quarta e última fase consiste em efetuar o pedido, passando a quantidade de peças a ser produzida. Esta ação dispara a produção das peças.

4.4 Considerações, Vantagens e Desvantagens

Os passos apresentados para criar um modelo no Automod são apenas os principais. Existem muitas outras atividades que devem ser feitas para tornar o modelo criado completamente ativo. Conclui-se então que a produção de novos modelos requer muito tempo de análise, preparação dos dados, projeto e execução da simulação, necessitando de projetistas altamente especializados. Além disso, os modelos construídos no Automod não permitem reuso, pois cada simulação foi feita especialmente para uma determinada realidade. Isto faz com que o tempo e o custo de simulação fiquem muito além das expectativas dos empresários, que necessitam de respostas rápidas e econômicas. Nenhuma empresa se proporia a esperar muito tempo para, por exemplo, colocar em funcionamento uma nova máquina ou linha de produção esperando algumas respostas da simulação. Isto porque as máquinas se depreciam e necessitam estar produzindo constantemente para retornar às empresas os valores investidos.

Por outro lado, a utilização da ferramenta Automod já provou ser bastante interessante em várias aplicações. Muitos projetos de simulação já foram desenvolvidos para grandes empresas como a General Motors, Volkswagen, Boticário e a Empresa Brasileira de Correios e Telégrafos (MSE MANUFACTURING STRATEGIES, 1999), resultando em importantes estudos. Ganhos de produção de até 40% foram simulados sem que se mudasse de lugar uma só máquina, ou um só homem fosse realocado. Com isso conclui-se que a simulação é, sem dúvida, uma importante metodologia de análise e projeto e o Automod uma eficiente ferramenta, com o mesmo objetivo.

Avaliando o que foi dito acima, é visível que o uso da simulação e do Automod como ferramenta de modelagem só é recomendado em situações muito específicas que dispõe de muito tempo e dinheiro para criar um ambiente simulado que possa fornecer respostas precisas sobre novas modificações ou implantações nos sistemas de manufatura.

Fica evidente, depois de analisar as duas metodologias de simulação apresentadas, que a utilização de uma biblioteca de classes orientada a objetos oferece muitas vantagens, se comparada com a metodologia de construção do Automod, que são citadas abaixo:

- **Facilidade de construção de modelos**: através da biblioteca de classes apresentada pode-se construir um modelo muito mais facilmente devido ao fato de que os conceitos relevantes da realidade são diretamente modelados na aplicação;
- **Rapidez na construção de modelos**: acompanhando o processo de modelagem em ambas as metodologias, fica claro que a orientação a objetos agiliza o processo de simulação, permitindo que os modelos possam ser construídos rapidamente adequando-se às exigências das empresas com relação a tempo e custos;
- **Reuso de modelos**: a biblioteca de classes herda da orientação a objetos uma característica muito importante que é o reuso. Através de classes da biblioteca como a classe `Celula_Manufatura`, é possível criar componentes pré-definidos de recursos físicos que podem ser usados em vários modelos. Estes componentes permitem aumentar ainda mais a velocidade de modelagem, já que oferecem estruturas que já foram previamente testadas.
- **Possibilidade de integração com o Automod**: através de um mapeamento entre as classes da biblioteca e as entidades do Automod, pode ser possível integrar as características de rápida prototipação, reuso e facilidade de construção com a interface gráfica e a infraestrutura de simulação do automod. Este estudo será proposto mais adiante, no capítulo 5.

4.4.1 Estudo Comparativo com outras Metodologias

O estado da arte na área de simulação em sistemas de manufatura apresenta vários esforços de pesquisa voltados para a criação de sistemas flexíveis de manufatura e novas abordagens na modelagem e simulação destes ambientes.

Comparando a metodologia baseada na hierarquia de classes apresentada nesta dissertação com alguns esforços relevantes neste sentido apresentados por Narayanan, pode-se perceber que este trabalho aproxima-se mais com a hierarquia BLOCS/M, da

Universidade de Berkeley. Tanto a definição das classes físicas, como as especificações do comportamento são bastante similares.

De todas as pesquisas apresentadas na análise do estado da arte, com certeza a que se aproxima mais do que está sendo proposto nesta dissertação é o trabalho de Park. Naquele trabalho, o autor propõe uma metodologia de modelagem chamada JR-net, que usa idéias da orientação a objetos e de redes de petri. O objetivo daquele trabalho que é o de usar esta metodologia de modelagem como *front-end* para simuladores de sistemas de manufatura comerciais como o Automod, é muito parecido com a proposta desta dissertação.

Ambos os trabalhos visam facilitar e agilizar a construção de modelos de sistemas de manufatura através da orientação a objetos. Entretanto, este trabalho apresenta duas principais vantagens sobre aquele:

A primeira vantagem é a maior possibilidade de se estender classes, possibilitando criar recursos extremamente especializados, já que é apresentada uma biblioteca de classes com os recursos bem definidos. No trabalho de Park, são propostos grupos de recursos com características similares, formando fragmentos padronizados que dificultam a extensão.

A segunda vantagem deste trabalho é a abordagem usada para controlar os recursos físicos. Esta abordagem baseia-se em estruturas bastante conhecidas em praticamente todos os ambientes de manufatura que são os roteiros de produção. Os roteiros de produção possuem as atividades que especificam o que e aonde devem ser realizadas operações sobre as peças. Esta abordagem agiliza a construção de novos modelos já que os relacionamentos entre as atividades e os recursos estão especificados explicitamente no roteiro de produção.

4.4.2 Resultados Quantitativos e Comparações

Nesta seção serão apresentados os resultados obtidos através da simulação em ambas as metodologias apresentadas, comparando-as com o roteiro original da empresa. O tempo total de produção da peça 15228 na Célula de Ponteiras é de 3480 segundos. Como o roteiro não leva em consideração o tempo de translado das peças, é necessário adicionar ao tempo total de produção este tempo. Assim, o tempo total é de 3615 segundos. Resumindo, temos:

Tempo de Produção		3480 segundos
Tempo de Translado	+	135 segundos
Tempo Total		3615 segundos

Para cada metodologia de simulação, foram executadas 20 simulações, sob as mesmas condições de teste. Segundo informações da equipe de produção da empresa (MARIANI, 2001), o tempo de produção real de cada atividade não difere em mais de 10% do valor especificado no roteiro. Assim, tanto no Automod, quanto na simulação em C++ foram utilizadas distribuições normais configuradas para randomizar números baseados no tempo padrão das atividades que estão na faixa de 10% para mais ou para menos.

Na tabela 2 e figura 21 são apresentados os resultados da simulação no Automod.

TABELA 2 – Resultados da simulação no Automod

Amostra	Tempo Simulado
1	3606
2	3749
3	3708
4	3809
5	3860
6	3697
7	3807
8	3898
9	3956
10	3622
11	3629
12	3619
13	3758
14	3839
15	3974
16	3711
17	3754
18	3498
19	3999
20	3731

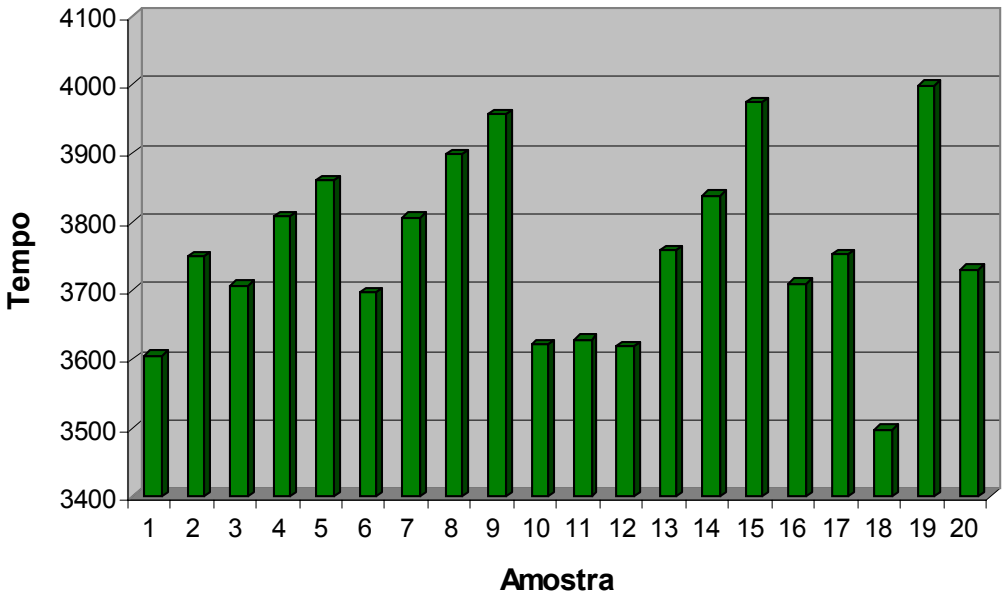

FIGURA 21 – Gráfico da simulação no Automod

Na tabela 3 e figura 22 são apresentados os resultados da simulação usando a biblioteca de classes implementada em C++.

TABELA 3 – Resultado da simulação em C++

Amostra	Tempo Simulado
1	3594
2	3778
3	3904
4	3897
5	3693
6	3851
7	3732
8	3778
9	3774
10	3785
11	3675
12	3711
13	3923
14	3682
15	3801
16	4010
17	3757
18	3603
19	3725
20	3874

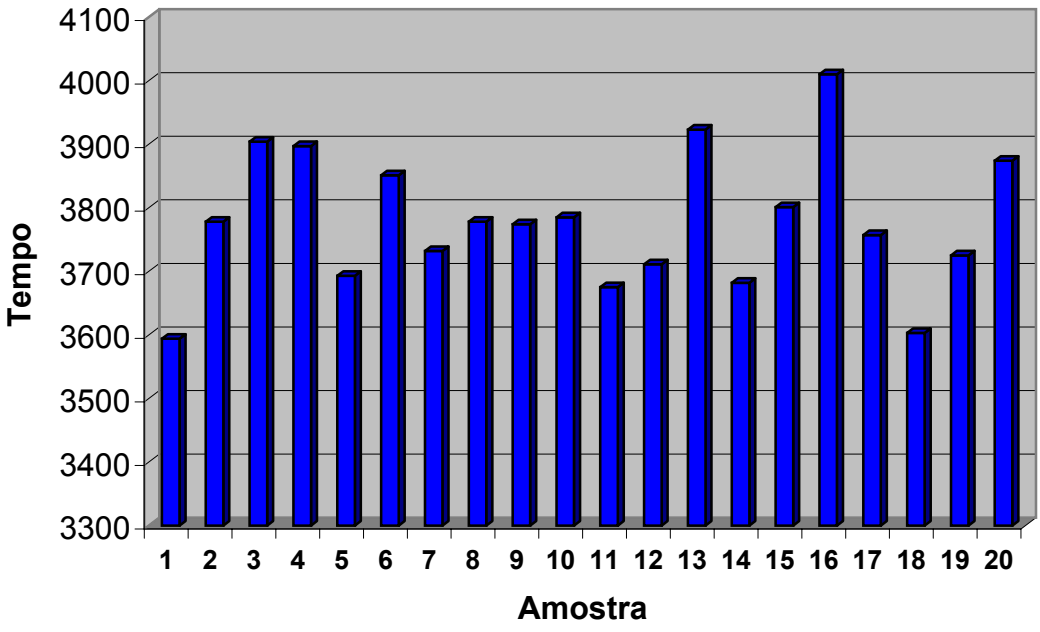

FIGURA 22 – Gráfico da simulação C++

Na tabela 4 e na figura 23 pode-se comparar as médias aritméticas de ambas as simulações com o tempo total especificado no roteiro de produção.

TABELA 4 – Comparação entre as médias aritméticas das simulações

Tempo do Roteiro	Média Automod	Média C++
3615	3761	3777

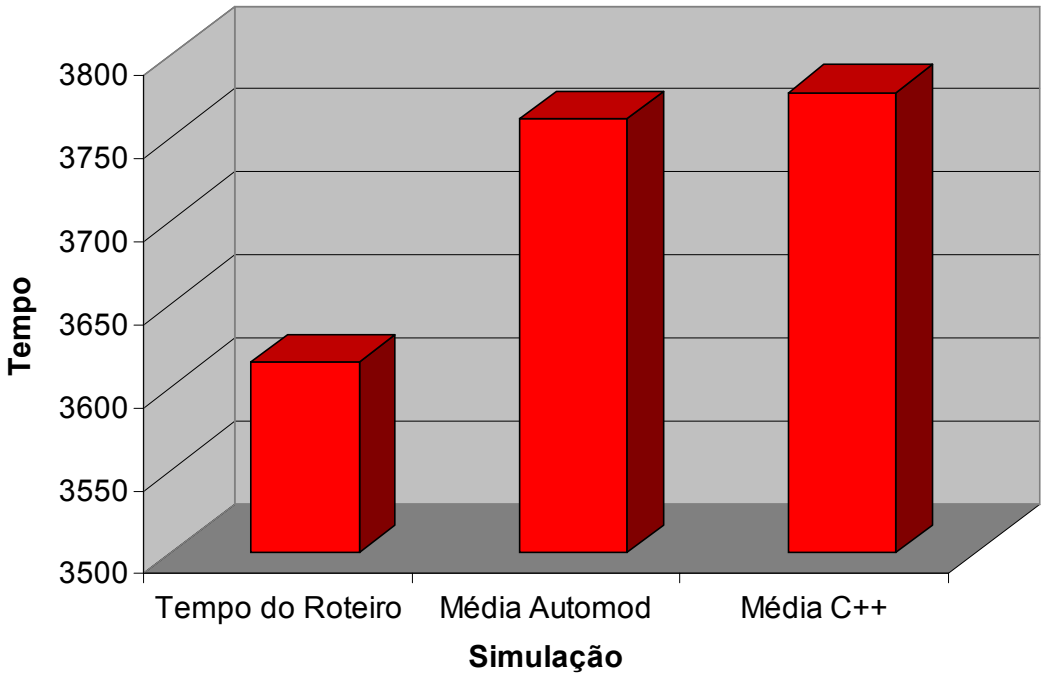

FIGURA 23 – Gráfico das médias dos resultados das simulações

Para garantir que os resultados obtidos com ambas as simulações são confiáveis, são apresentadas algumas medidas estatísticas como o desvio padrão, a variância e principalmente o intervalo de confiança. Segundo Downing (1998), o intervalo de confiança é um intervalo baseado em observações de uma amostra e construído de maneira que haja uma probabilidade especificada de o intervalo conter o verdadeiro valor desconhecido de um parâmetro. Para isso foi usado um nível de confiança de 95%. Este valor indica a probabilidade de o intervalo calculado conter o verdadeiro valor do parâmetro.

A tabela 5 apresenta as medidas estatísticas calculadas para ambas as simulações. Na figura 24, o gráfico mostra a média e o intervalo de confiança das simulações e o tempo de produção especificado no roteiro de produção, indicando que este valor está dentro do intervalo de confiança das simulações.

TABELA 5 – Medidas estatísticas de ambas as simulações

Medida	Automod	C++
Variância	16986	11060
Desvio Padrão	130	105
Intervalo de Confiança	[3505, 4016]	[3570, 3983]

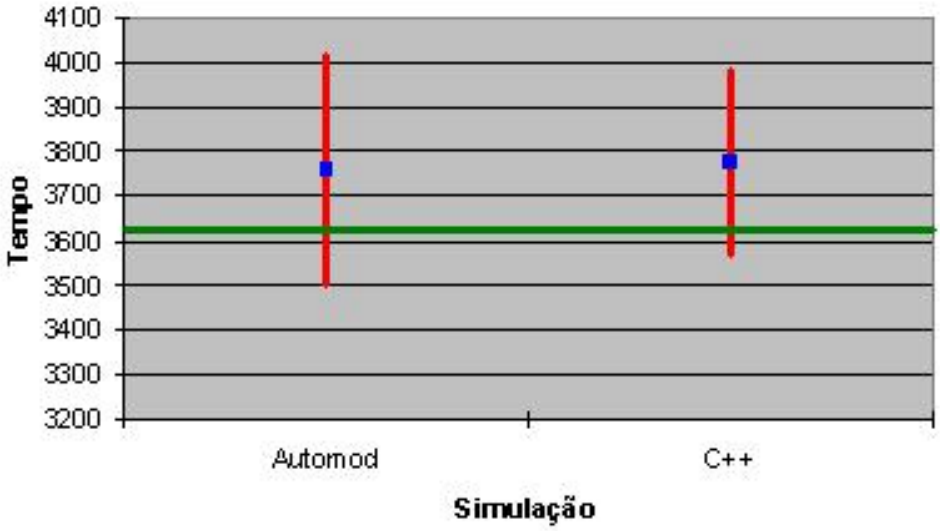

FIGURA 24 – Gráfico dos intervalos de confiança de ambas as simulações

Analisando os dados apresentados pode-se notar que a diferença entre o tempo do roteiro não difere em mais de 5% em ambas as simulações. A diferença nas simulações se deve ao fato de que o tempo de passagem no sistema é computado desde o ingresso da peça na fila de entrada. Se a peça demora um pouco mais para sair da fila, seu tempo de permanência na célula também tende a aumentar. Além do tempo de permanência da peça na simulação, os tempos de translado entre uma máquina e outra foram adicionados ao tempo do roteiro de produção seguindo uma estimativa. Para gerar um número mais correto, seria necessário um estudo específico com coleta de tempos na própria célula de manufatura. Se estes tempos tivessem sido considerados de forma mais precisa, sem dúvida os resultados de ambas as simulações tenderiam mais para o valor constante no roteiro de produção. Entretanto, esta análise extra exigiria muito tempo e acrescentaria pouco ao objetivo final do trabalho.

Outro dado importante que deve ser levado em consideração é o fato de que ambas as simulações partiram do mesmo estudo de caso, usando os mesmos dados de entrada, mas foram modelados em momentos diferentes. Como os dados obtidos apresentaram-se bastante parecidos, pode-se concluir que ambas metodologias apresentam grande exatidão sobre os dados de entrada submetidos e pode-se considerar validada a biblioteca de classes apresentada para este tipo de aplicação.

5 Proposta de Integração

O Automod, como já comentado, é uma ferramenta de modelagem e simulação bastante poderosa. A interface gráfica de modelagem e sua linguagem de programação proprietária permitem que o projetista obtenha resultados bastante interessantes no que se refere a projetos de novos sistemas de manufatura, novas linhas de produção, testes de novos leiautes de chão de fábrica e posicionamento de novos recursos. Apesar de suas características, a construção de modelos no Automod deve ser feita por pessoas capacitadas e que tenham noções aprofundadas em várias áreas do conhecimento humano como a teoria de manufatura, simulação, programação e administração. Além disso, o tempo de análise e modelagem da simulação tornam-se muito longos, o que inviabiliza muitas das aplicações potenciais.

A situação ideal seria a construção do modelo usando a abordagem orientada a objetos, através da biblioteca de classes apresentada neste trabalho. A partir do modelo construído, e utilizando-se uma ferramenta de tradução, este modelo poderia ser carregado no Automod de modo a utilizar todas suas ferramentas de simulação de manufatura. Neste caso, a biblioteca orientada a objetos poderia ser classificada como uma ferramenta de modelagem rápida de simulações de manufatura que seria um *front-end* para o Automod. Desta maneira, os modelos construídos podem adquirir qualidades de ambas as metodologias. Do Automod obtêm-se uma interface gráfica em três dimensões com várias opções de contadores, distribuições, controle da simulação, etc. Da biblioteca de classes, temos a rápida modelagem, o reuso e uma metodologia conhecida universalmente.

5.1 Mapeamento do Modelo de Classes para o Automod

Nesta seção será proposto a nível conceitual o mapeamento das classes construídas na biblioteca de classes em entidades de simulação no Automod. O mapeamento das classes para o Automod é o primeiro passo na tentativa de integração entre as duas abordagens. Havendo classes equivalentes a grande parte das estruturas do Automod, torna-se possível a integração.

Analisando as estruturas da biblioteca de classes e do Automod, não há dúvida de que o mapeamento deve iniciar a partir dos recursos físicos, tanto de manufatura, quanto de transporte. No caso de recursos de manufatura, cada instância do modelo orientado a objetos equivale a uma entidade do tipo recurso no Automod, que pode ser configurado através da opção *Resources* do menu *Process*. Esta tradução é bastante simples já que a equivalência um para um se configura neste caso. A posição física no modelo pode ser obtida através de uma instância obrigatória relacionada à classe Recurso chamada `Posicao` e fornece os valores para o posicionamento das máquinas. Junto com a especificação de cada recurso, pode-se configurar uma fila de peças, representada no diagrama de classes pela classe `Fila_Pecas`. Para criar uma fila para cada recurso, deve-se usar a opção *Queues*, no menu *Process*, configurando o parâmetro *Default Capacity* conforme a capacidade da fila.

O mapeamento dos recursos de transporte também é obtido de forma similar aos recursos de manufatura. A diferença está no fato de que não pode ser usada a equivalência um para um neste caso. No Automod os recursos de transporte são mais complexos do que as máquinas. Para especificar um transportador do tipo esteira, por exemplo, deve-se criar um sistema através do menu *System*, opção *Conveyor*. Um menu aparece na tela permitindo ao projetista desenhar todos os segmentos que fazem parte deste sistema de esteira. No diagrama de classes, temos a classe `Transporte` e suas especializações `Robo_Transporte`, `Esteira` e `AGV`. Cada instância de uma destas classes ao ser mapeada para o Automod deve criar um novo sistema de transporte. Cada instância da classe `Segmentos` cria uma parte deste sistema de transporte em particular. Se o sistema for um AGV, deve-se ainda mapear a entidade transportadora que foi instanciada no diagrama de classes. As entidades transportadoras nos AGV's são correspondentes aos veículos no Automod.

Das classes que controlam a produção, nem todas possuem entidades equivalentes no Automod, porque o mesmo não leva em conta a estrutura do produto e sim o processo de manufatura. Desta forma, as classes `Plano_Mestre`, `Capacidade_Diaria`, `Algoritmo_Escalonamento`, `Pedido` e `Ordem_Producao` não podem ser mapeadas. Isto significa que a tradução do modelo de classes para o Automod irá apresentar uma perda de funcionalidade. Verifica-se então que a simulação de sistemas usando a biblioteca de classes apresentada possui mais possibilidades de modelagem do que o Automod.

Para especificar a seqüência de operações das peças, devem ser usadas as instâncias da classe `Atividade`. Como as atividades estão relacionadas a uma peça, cada peça vai gerar um conjunto de processos. Cada atividade gera um processo no Automod. Os processos são criados através do menu *Process System*, opção *Process*. O relacionamento de cada atividade com sua respectiva próxima atividade vai gerar uma configuração no Automod que indica o próximo processo. Na configuração de cada processo, na janela *Process*, existe uma opção denominada *Default Next Process*, que deve ser configurada usando a próxima atividade. Se não houver uma próxima atividade, coloca-se na opção *Default Next Process* a palavra *Die*, indicando que a peça deve desaparecer depois de passar pelo último processo.

A parte mais complexa no mapeamento é a geração do código fonte na linguagem proprietária do Automod. Para especificar o comportamento de cada processo, é necessário criar procedimentos chamados *Arriving Procedures* ligados a cada processo que indicam o caminho seguido pela peça em cada processo. Nestes procedimentos deve-se especificar onde a peça vai ingressar, o tempo que a peça irá ocupar o recurso, o caminho seguido por ela até o próximo recurso e o próximo processo. A configuração de onde a peça irá ingressar está especificado no relacionamento entre as atividades e os equipamentos no diagrama de classes. O tempo que a peça irá alocar o recurso pode ser obtido no atributo `Tempo_padrao` da classe `Atividade`. O relacionamento `prox_atividade` da classe atividade gera a informação que especifica o próximo processo. Para configurar o caminho a ser seguido pela peça até o próximo processo, deve existir um algoritmo que realize uma consulta às classes agregadas à classe `Estrutura_Transporte`. Isto permitirá que se descubra em qual ponto de conexão a peça deve entrar, qual deve sair e por qual recurso a mesma deve seguir.

5.2 Ferramenta de Tradução

Tendo-se definido o mapeamento das classes da biblioteca em entidades do Automod, seria apropriada a criação de uma ferramenta de tradução automática entre as duas abordagens. Isto iria agilizar sobremaneira a construção de simulações, retirando completamente a necessidade de um especialista no Automod. A primeira função importante que esta ferramenta deveria ter é um gerador automático de atividades e equipamentos, inclusive especificando os relacionamentos entre eles. Isto tudo pode ser obtido a partir de um roteiro de produção qualquer que serve de código fonte de entrada para um interpretador que seleciona a peça, as atividades correspondentes, os equipamentos e os relacionamentos entre estes. Todas estas informações podem ser obtidas facilmente nos roteiros de produção das fábricas atuais.

A função mais importante da ferramenta de tradução é a de transformar um modelo orientado a objetos construído usando as classes da biblioteca de classes proposta neste trabalho em uma simulação no Automod. Para isso seria necessário o estudo aprofundado dos arquivos de configuração do Automod de modo a descobrir a construção interna destes arquivos. De posse desta estrutura, o tradutor poderia mapear automaticamente as classes do modelo para os arquivos de configuração do Automod. Feito isso, seria só gerar o modelo a partir de sua compilação, usando o compilador do Automod, que produziria uma simulação perfeitamente funcional.

5.3 Exemplo de Mapeamento

Mesmo tendo definindo resumidamente o modo de mapeamento entre as duas metodologias apresentadas, é por meio de um exemplo prático que estas definições se tornam mais claras. Pode-se utilizar o estudo de caso apresentado neste trabalho para demonstrar praticamente como se daria o mapeamento entre as metodologias. O diagrama de instâncias construído a partir da biblioteca de classes para representar o estudo de caso será usado para demonstrar o mapeamento. Também será usada a simulação do estudo de caso criada no Automod. Como se trata do mesmo caso, a estrutura usada em ambas as metodologias possuem designações similares, deixando mais claro o exemplo.

A primeira fase do mapeamento, como já citado em seções anteriores, é a tradução dos recursos físicos, tanto de transformação, como de transporte. No exemplo da célula de manufatura de pontas de eixo da Pigozzi têm-se 8 recursos de manufatura e 4 recursos de transporte, como se pode visualizar no diagrama de instância apresentado no Anexo 2. Estes recursos instanciados através das classes correspondentes da biblioteca podem ser mapeados para o Automod facilmente. No menu *Process*, opção *Resources* são cadastrados os vários recursos de manufatura, configurando seus atributos. Sua posição física também é definida nesta janela através do botão *Edit Graphic*, usando os dados da instância da classe `Posicao`. A figura 25 apresenta as classes instanciadas da biblioteca e os locais correspondentes de configuração no Automod. O nome da instância corresponde ao nome do recurso no Automod. Os atributos `Cod_Recurso` e `Estado` não são usados no mapeamento. O atributo `Tempo_Preparacao` deve ser configurado através da inserção de um ciclo do recurso na janela *Attached Resource Cicle*. Esta opção permite a definição de ações que ocorrem em ciclos como o tempo de preparação e o tempo médio entre falhas (MTBF).

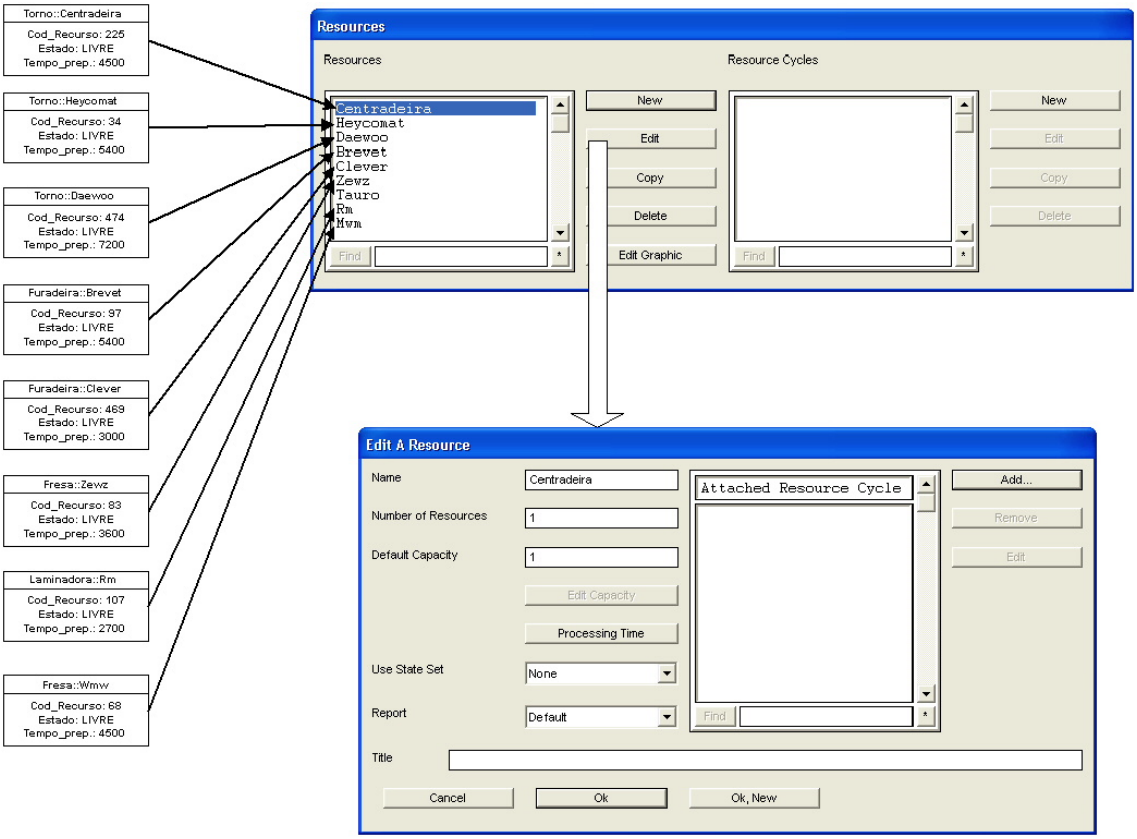

FIGURA 25 – Mapeamento das instâncias de recursos de manufatura para o Automod

Os recursos de transporte são mapeados de forma diferente dos recursos de manufatura. O Automod trata cada recurso de transporte como um processo separado. Cada instância de um recurso de transporte deve criar um novo sistema no Automod. Um novo sistema é criado através do menu *System*, opção *New*. Depois de criado o novo sistema, uma janela de ferramentas de desenho é mostrada permitindo ao projetista desenhar o sistema de transporte. Na biblioteca de classes, o desenho do sistema de transporte é especificado através da classe `Segmentos`. No exemplo, existem 4 instâncias da classe AGV. Cada uma destas instâncias deve ser transformada em um sistema no Automod, como mostra a figura 26. Como o sistema de transporte é um AGV, ainda é necessário configurar o veículo que circulará pelo caminho. A figura 26 mostra também o mapeamento do transportador para o Automod. O nome da instância gera o nome do sistema. O atributo `Tipo` da classe `Transportador` mapeia a especificação do nome do veículo, no caso, um homem. O atributo capacidade é configurado na janela *Edit a Vehicle Definition*, no campo *Vehicle Capacity*. O atributo velocidade é configurado na janela *Specifications by Load Type*, onde cada tipo de carga pode ter configurações de velocidade e aceleração únicas.

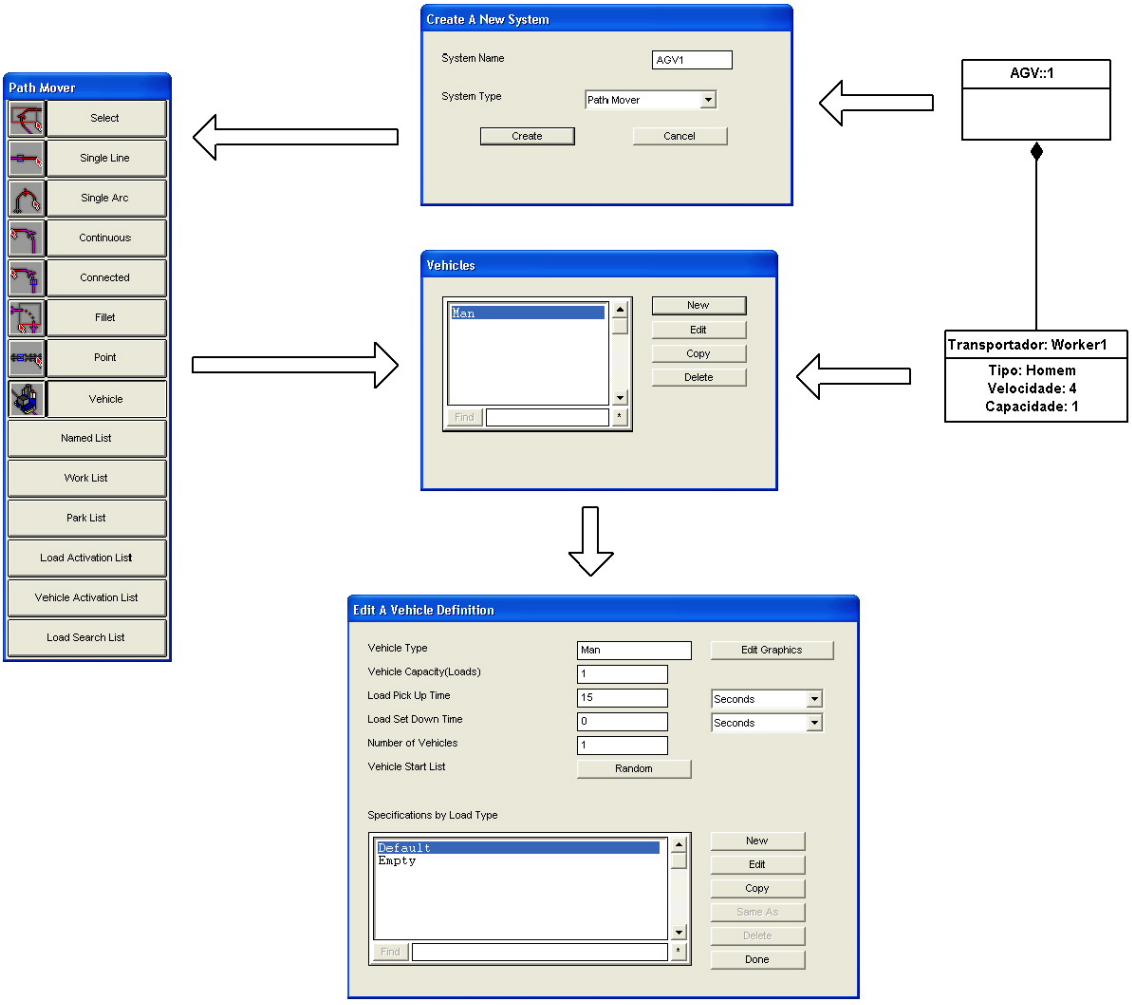

FIGURA 26 – Mapeamento das instâncias de recursos de transporte no Automod

Pelo fato do Automod ser um simulador orientado a processos, o modo de controlar a simulação se dá através de processos. Em cada processo existem instruções precisas das ações que a peça deve executar durante sua passagem pelo processo em questão. A estrutura equivalente aos processos do Automod são as atividades na biblioteca de classes. Para mapear as instâncias das atividades em processos no Automod, é necessário simplesmente cadastrar os novos processos no menu *Process*, opção *Process*. Este mapeamento pode ser visto na figura 27. No exemplo, cada atividade mapeia um processo, que no caso está nomeado com o designativo do recurso onde a atividade é executada. O atributo `Cod_Atividade` não é utilizado. O atributo `Descricao` pode gerar o nome do processo. O atributo `Tempo_Padrao` não pode ser mapeado diretamente em alguma estrutura do Automod. Ele deve ser inserido dentro de um procedimento chamado *Arriving Procedure*, acessível através do botão *Arriving Procedure* da janela *Edit a Process*, mostrada na figura 27. Este procedimento é responsável por especificar as ações que a peça deve executar enquanto encontra-se neste processo. Este procedimento é escrito em uma linguagem proprietária do Automod. Um trecho de código desta linguagem pode ser visto na figura 16 do capítulo anterior. Para criar estes trechos de procedimento, seria necessária, com já comentado, a criação de uma rotina de tradução que seria responsável por interpretar os caminhos que as peças seguem, podendo gerar o procedimento *Arriving Procedure* automaticamente. Provavelmente, o atributo `Tempo_Padrao` seria um parâmetro de entrada desta rotina.

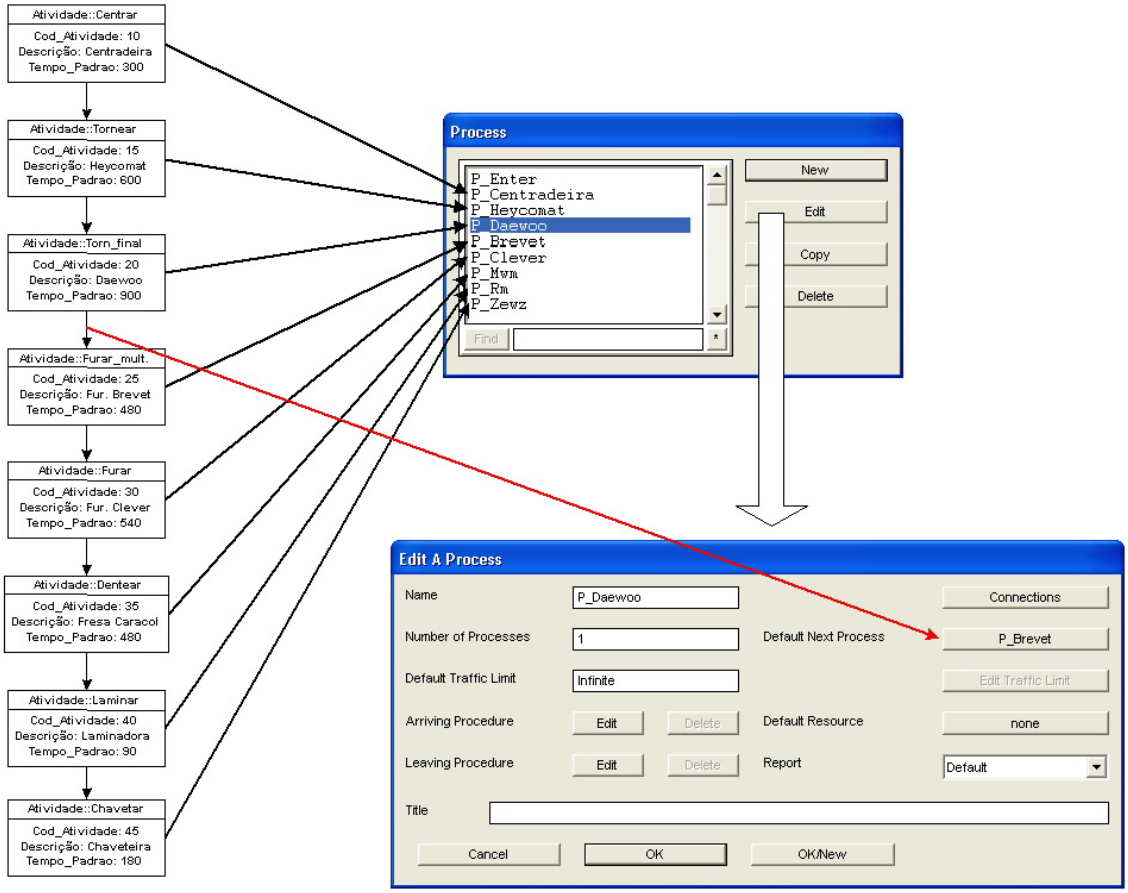

FIGURA 27 – Mapeamento das instâncias de atividades em processos no Automod

6 Conclusão e Trabalhos Futuros

O estudo realizado no início do desenvolvimento deste trabalho gerou o senso comum que apontou para a especificação de um esquema de modelagem para sistemas de manufatura que permitisse preencher as lacunas deixadas por um sem número de esforços neste sentido, mas que se voltam para aplicações muito específicas. Este fato gerou a motivação necessária para a proposição da biblioteca de classes para sistemas de manufatura apresentada neste trabalho. Por todo o contexto apresentado, pôde-se concluir que a metodologia orientada a objetos é adequada para a modelagem e simulação em sistemas de manufatura, pelo fato dos objetos conseguirem representar acuradamente e facilmente as entidades do mundo real.

O formato final da biblioteca de classes permite capturar as principais definições dos sistemas de manufatura reais (indústria de transformação), como o uso dos roteiros de produção e atividades, facilitando e agilizando a modelagem destes sistemas. A biblioteca utiliza as características da orientação a objetos, encorajando o reuso e aumentando a produtividade na construção de modelos. Além disso, ela implementa um esquema de construção da infra-estrutura de transporte que utiliza definições das redes de computadores, permitindo que o projetista possa criar sistemas de deslocamento de peças complexos e de acordo com o nível de abstração necessário.

Para gerar resultados conclusivos sobre a validade da biblioteca de classes apresentada, foi usado um estudo de caso. Este estudo de caso, baseado em uma célula de manufatura de pontas de eixo da empresa Pigozzi S/A, foi modelado usando as classes da biblioteca proposta. Como a biblioteca foi proposta de forma conceitual, seria necessária a implementação das classes da biblioteca em uma linguagem de programação orientada a objetos. Usando a linguagem C++, grande parte das classes da biblioteca foram implementadas, podendo assim gerar um modelo do estudo de caso, que foi apresentado na seção 4.3. Com este estudo pôde-se notar o quão fácil e rápida fica a modelagem dos sistemas de manufatura. Simplesmente definindo as instâncias, os atributos e os relacionamentos, o modelo fica completo e pode ser simulado, gerando resultados em muito pouco tempo. Estes resultados foram bastante precisos e ficaram dentro do intervalo de erro de 5% se comparado ao tempo de produção real da célula de manufatura.

O estudo para a concepção da biblioteca de classes apresentada partiu de uma estrutura de classes genérica, formada por classes abstratas que podem ser utilizadas em qualquer sistema de manufatura. O conjunto genérico foi especializado e ampliado com a realização do estudo de caso. Uma vez que a aplicação escolhida é fortemente baseada nos processos de transformação, a biblioteca final é mais completa para a modelagem deste grupo de sistemas de manufatura.

Apesar do estudo de caso modelado com o auxílio da biblioteca de classes ter apresentado resultados bastante interessantes, seria necessário um parâmetro de comparação para a validação da biblioteca. Para isso, foi gerado um modelo do mesmo estudo de caso no simulador Automod. O modelo construído no Automod gerou resultados igualmente precisos, permitindo inclusive definir informações importantes

sobre a realidade modelada, como os recursos gargalo. Entretanto, o tempo de modelagem no Automod foi muito maior do que o conseguido com o auxílio da biblioteca de classes proposta. Este fato é a principal explicação para a pequena utilização da simulação nas empresas atualmente. Os empresários geralmente não podem esperar pelos resultados da simulação, pois as empresas necessitam de respostas rápidas para competir em um mercado cada vez mais acirrado.

Por fim, foi apresentada uma proposta de mapeamento das classes da biblioteca para entidades do Automod, com o objetivo de fazer com que a hierarquia apresentada comporte-se como um *front-end* para o Automod. A conclusão obtida desta proposta é de que esta integração é perfeitamente possível e muito benéfica, já que pode permitir a integração das funcionalidades de ambas as metodologias. O aumento da velocidade e a facilidade de construção podem vir a incentivar o uso da simulação em outras aplicações atualmente inviáveis.

Como trabalhos futuros pode-se usar a proposta de integração apresentada neste trabalho e implementar uma ferramenta de tradução que permita, através de um modelo criado usando a biblioteca de classes, obter um modelo funcional no Automod.

Uma possível extensão deste trabalho é o desenvolvimento de classes que permitam a modelagem de sistemas de manufatura voltados a indústrias de montagem. A partir desta idéia, pode-se criar *frameworks* de classes voltados a uma aplicação específica, como a indústria de transformação, petroquímica e de montagem. Além disso, uma ampliação dos estudos de caso seria necessária para a correta validação de todas as extensões.

Outra possibilidade é a de estender a ferramenta SIMOO-RT de forma que esta integre completamente a ferramenta de tradução da biblioteca de classes, gerando modelos completos no Automod e podendo gerar código em linguagens de tempo real, como o AO/C++ (PEREIRA, 1994) ou RT-Java.

Anexo 1 Roteiro de Produção 15228

PIGOZZI S/A - ENGRENAGENS E TRANSMISSOES DATA: 16/11/01 PAGINA - 1
PLANEJAMENTO INDUSTRIAL SEQUENCIA OPERACIONAL

ATENCAO: O uso dos EPI'S fazem parte das operações

DESCRICAO: PONTA DE EIXO 365 NRO. PECA: 015228-5
NRO. ROTEIRO: 015228 ULTIMA ALTERACAO: 13/06/01 COD.LIB: 00 DIVISAO: 00
DATA: 17/12/98 ELABORADO POR ELOY SIQUEIRA BORGES NRO. ALTERACOES: 7

SEQ. OPER.: 005 DESCRICAO: DESVIO DE PROCESSO USIN
TEMPO OPERACAO: 0,00 TEMPO PREPARO: 0,00 MAQUINA: 000000 DISP: 000000

SEQ. OPER.: 010 DESCRICAO: CENTRADEIRA USIN
TEMPO OPERACAO: 5,00 TEMPO PREPARO: 75,00 MAQUINA: 000225 DISP: 000000

SEQ. OPER.: 015 DESCRICAO: TORNO COP.HEYCOMAT AUT.3/750 USIN
TEMPO OPERACAO: 10,00 TEMPO PREPARO: 90,00 MAQUINA: 000034 DISP: 000000

SEQ. OPER.: 020 DESCRICAO: TORNO HORIZ. CNC DAEWOO PUMA 350B USIN
TEMPO OPERACAO: 9,68 TEMPO PREPARO: 120,00 MAQUINA: 000172 DISP: 000000

SEQ. OPER.: 025 DESCRICAO: FUR.MULT.BREVET FU.310N172 102 USIN
TEMPO OPERACAO: 8,00 TEMPO PREPARO: 90,00 MAQUINA: 000097 DISP: 000000

SEQ. OPER.: 030 DESCRICAO: FURADEIRA RADIAL CLEVER Z3050X16 USIN
TEMPO OPERACAO: 9,00 TEMPO PREPARO: 50,00 MAQUINA: 000469 DISP: 000000

SEQ. OPER.: 035 DESCRICAO: FRESAD.CARACOL MOR.ZEWZ 500X5 USIN
TEMPO OPERACAO: 8,00 TEMPO PREPARO: 60,00 MAQUINA: 000083 DISP: 000000

SEQ. OPER.: 040 DESCRICAO: LAMINADORA DE ROSCA MOD.RM 54A USIN
TEMPO OPERACAO: 1,50 TEMPO PREPARO: 45,00 MAQUINA: 000107 DISP: 000000

SEQ. OPER.: 045 DESCRICAO: FRESA CHAVETA WMW USIN
TEMPO OPERACAO: 3,00 TEMPO PREPARO: 75,00 MAQUINA: 000068 DISP: 000000

SEQ. OPER.: 050 DESCRICAO: DESVIO DE PROCESSO USIN
TEMPO OPERACAO: 0,00 TEMPO PREPARO: 0,00 MAQUINA: 000000 DISP: 000000

F I M D E R O T E I R O

TEMPO TOTAL DE PREPARO: 755,00 TEMPO TOTAL DE OPERACAO: 54,18

Anexo 2 Diagrama de Instâncias do Roteiro 15228

Anexo 3 Código C++ da Implementação das Classes da Biblioteca

Classe Area_Armazenamento

- Area_Armazenamento.h

```
#ifndef ARMAZENAMENTO_H
#define ARMAZENAMENTO_H

#include "Equipamento.h"

// Classe derivada da classe Equipamento
class Area_Armazenamento : public Equipamento {
public:
      Area_Armazenamento(int, int, int, char*, int, int); //construtor
      virtual ~Area_Armazenamento();           // destrutor
      int getCapacidade();
      int getAtual();
      bool adiciona();
      bool retira();

private:
      int Capacidade;
      int Atual;
};

#endif
```

- Area_Armazenamento.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Area_Armazenamento.h"

// Construtor para a classe Area_Armazenamento
Area_Armazenamento::Area_Armazenamento(int cod, int est, int prep,
char t[30], int capa, int atu)
      :Equipamento (cod, est, prep, t)
{
      Capacidade = capa;
      Atual = atu;
}

Area_Armazenamento::~Area_Armazenamento()
{     }

int Area_Armazenamento::getCapacidade()
{
      return Capacidade;
```

```
}

int Area_Armazenamento::getAtual()
{
	return Atual;
}


// Adiciona uma peca na Area de armazenamento
bool Area_Armazenamento::adiciona()
{
	if (Capacidade == Atual) {		// Testa se area esta cheia
		cout << "Area de Armazenamento cheia" << endl;
		return false;
	}
	else {
		Atual++;
		return true;
	}
}

// Retira uma peca da Area de armazenamento
bool Area_Armazenamento::retira()
{
	if (Atual == 0) {		// Testa se area esta vazia
		cout << "Area de Armazenamento vazia" << endl;
		return false;
	}
	else {
		Atual--;
		return true;
	}
}
```

Classe `Atividade`

- Atividade.h

```
#ifndef ATIVIDADE_H
#define ATIVIDADE_H

#include "Equipamento.h"
#include "Estrutura_Transporte.h"

class Atividade {
public:
	Atividade(int, char*, int);		// Construtor
	virtual ~Atividade();			// Destrutor
	bool executaAtividade(bool);
	void criaRelacionamentoEquip(Equipamento*);
	int getCod_Atividade();
	char* getDescricao();
	int getTempo_padrao();
	void criaRelacionamentoTransp(Estrutura_Transporte*);

private:
	int Cod_Atividade;
	char Descricao[100];
	int Tempo_padrao;
```

```
      Equipamento *refEquipamento;
      Estrutura_Transporte *refTransporte;
};

#endif
```

- Atividade.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Atividade.h"

// Construtor da classe Atividade
Atividade::Atividade(int cod, char des[100], int tempo)
{
      Cod_Atividade = cod;
      strcpy(Descricao, des);
      Tempo_padrao = tempo;
}

// Destrutor
Atividade::~Atividade()
{     }

void Atividade::criaRelacionamentoEquip(Equipamento *equip)
{
      refEquipamento = equip;
}

void Atividade::criaRelacionamentoTransp(Estrutura_Transporte *trans)
{
      refTransporte = trans;
}

int Atividade::getCod_Atividade()
{
      return Cod_Atividade;
}

char* Atividade::getDescricao()
{
      return Descricao;
}

int Atividade::getTempo_padrao()
{
      return Tempo_padrao;
}

bool Atividade::executaAtividade(bool ultima)
{
      bool ativ = true;
      bool trans = true;

      cout << "Alocando recurso: ";
      cout << refEquipamento->getCod_recurso();
      cout << "         Tipo: ";
```

```
      cout << refEquipamento->getTipo() << endl;;
      ativ = refEquipamento->aloca_recurso(Tempo_padrao);
      if (!ultima)
            refTransporte->translado();

      return ativ;
}
```

Classe Entidade_Simulada

- Entidade_Simulada.h

```
#ifndef ENTIDADE_H
#define ENTIDADE_H

#include "Relogio.h"

class Entidade_Simulada {
public:
      Entidade_Simulada();                // Construtor
      virtual ~Entidade_Simulada(); // Destrutor
      void criaRelacionamento(Relogio*);
      void simula(int);

private:
      Relogio *refRelogio;

};

#endif
```

- Entidade_Simulada.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Entidade_Simulada.h"
#include "Relogio.h"

// Definicao do construtor
Entidade_Simulada::Entidade_Simulada()
{     }

Entidade_Simulada::~Entidade_Simulada()       // Destrutor
{     }

void Entidade_Simulada::criaRelacionamento(Relogio *rel)
{
      refRelogio = rel;
}

void Entidade_Simulada::simula(int tempo)
{
      refRelogio->incrementaRelogio(tempo);
}
```

Classe `Equipamento`

- Equipamento.h

```
#ifndef EQUIPAMENTO_H
#define EQUIPAMENTO_H

#include "Recurso.h"

// Classe derivada da classe Recurso
class Equipamento : public Recurso {
public:
      Equipamento(int, int, int, char*);  // construtor
      virtual ~Equipamento();                   // destrutor
      int getTempo_preparacao();
      char* getTipo();

private:
      int Tempo_preparacao;
      char Tipo[50];
};

#endif
```

- Equipamento.cpp

```
#include <string>

#include "Equipamento.h"

// Construtor para a classe Equipamento
Equipamento::Equipamento(int cod, int est, int prep, char t[50])
      :Recurso(cod, est)
{
      Tempo_preparacao = prep;
      strcpy(Tipo, t);
}

Equipamento::~Equipamento()
{     }

int Equipamento::getTempo_preparacao()
{
      return Tempo_preparacao;
}

char* Equipamento::getTipo()
{
      return Tipo;
}
```

Classe Estrutura_Transporte

- Estrutura_Transporte.h

```
#ifndef ESTRUTURA_H
#define ESTRUTURA_H

#include "Entidade_Simulada.h"

class Estrutura_Transporte : public Entidade_Simulada {
public:
      Estrutura_Transporte(int);                // construtor
      virtual ~Estrutura_Transporte();    // destrutor
      int getTempotranslado();
      bool translado();

private:
      int Tempo_translado;
};

#endif
```

- Estrutura_Transporte.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using std::cout;
using std::endl;

#include "Estrutura_Transporte.h"
#include "Entidade_Simulada.h"

// Construtor da classe Esatrutura_Transporte
Estrutura_Transporte::Estrutura_Transporte(int trans)
      :Entidade_Simulada()
{
      Tempo_translado = trans;
}

// Destrutor
Estrutura_Transporte::~Estrutura_Transporte()
{     }

// Retorna Tempo_translado
int Estrutura_Transporte::getTempotranslado()
{
      return Tempo_translado;
}

bool Estrutura_Transporte::translado()
{
      int tempo;
      long t;

      srand(time(&t));
      cout << "Transportando peca" << endl;
      tempo = (Tempo_translado - (int)(Tempo_translado * 0.1));
      tempo = tempo + (rand() % (int)(Tempo_translado * 0.2));
```

```
	cout << "Tempo Total do Transporte: ";
	cout << tempo << endl;
	this->simula(tempo);
	cout << "Transporte concluido" << endl;

	return true;
}
```

Classe `Fresa`

- Fresa.h

```
#ifndef FRESA_H
#define FRESA_H

#include "Equipamento.h"

// Classe derivada da classe Equipamento
class Fresa : public Equipamento {
public:
	Fresa(int, int, int, char*, int);   // Construtor
	virtual ~Fresa();                          // Destrutor
	int getMagazine();

private:
	int Magazine;
};

#endif
```

- Fresa.cpp

```
#include "Fresa.h"

// Cosntrutor para a classe Fresa
Fresa::Fresa(int cod, int est, int prep, char t[30], int mag)
	:Equipamento (cod, est, prep, t)
{
	Magazine = mag;
}

// Destrutor
Fresa::~Fresa()
{	}

int Fresa::getMagazine()
{
	return Magazine;
}
```

Classe Furadeira

- Furadeira.h

```
#ifndef FURADEIRA_H
#define FURADEIRA_H

#include "Equipamento.h"

// Classe derivada da classe Equipamento
class Furadeira : public Equipamento {
public:
      Furadeira(int, int, int, char*, int);     // Construtor
      virtual ~Furadeira();                           // Destrutor
      int getRotacoes();

private:
      int Rotacoes;

};

#endif
```

- Furadeira.cpp

```
#include "Furadeira.h"

// Construtor da classe Furadeira
Furadeira::Furadeira(int cod, int est, int prep, char t[30], int rot)
:Equipamento (cod, est, prep, t)
{
      Rotacoes = rot;
}

// Destrutor
Furadeira::~Furadeira()
{     }

// Retorna o numero de rotacoes por minuto da furadeira
int Furadeira::getRotacoes()
{
      return Rotacoes;
}
```

Classe Laminadora

- Laminadora.h

```
#ifndef LAMINADORA_H
#define LAMINADORA_H

#include "Equipamento.h"

class Laminadora : public Equipamento {
public:
      Laminadora(int, int, int, char*, int);    // Construtor
      virtual ~Laminadora();                          // Destrutor
      int getDentes();
```

```
private:
      int Dentes;
};

#endif
```

- Laminadora.cpp

```
#include "Laminadora.h"

// Construtor para a classe Laminadora
Laminadora::Laminadora(int cod, int est, int prep, char t[30], int
den)
      :Equipamento(cod, est, prep, t)
{
      Dentes = den;
}

// Destrutor
Laminadora::~Laminadora()
{     }

// Retorna a quantidade de dentes da maquina
int Laminadora::getDentes()
{
      return Dentes;
}
```

Classe NodoListaAtividade

- NodoListaAtividade.h

```
#ifndef NODO_H
#define NODO_H

#include "Atividade.h"

class NodoListaAtividade {

public:
      NodoListaAtividade();
      virtual ~NodoListaAtividade();
      bool insereAtividade(Atividade*);
      Atividade* proximaAtividade(bool);
      bool ultimaAtividade();
      void voltaInicioLista();

private:
      Atividade *RefAtividade;
      NodoListaAtividade *ProximaAtividade;
      NodoListaAtividade *Header;
      NodoListaAtividade *Fim;
      NodoListaAtividade *Atual;

};

#endif
```

- NodoListaAtividade.cpp

```
using std::cout;
using std::endl;

#include "NodoListaAtividade.h"

// Construtor da classe NodoListaAtividade
NodoListaAtividade::NodoListaAtividade()
{
	Header = NULL;
	Atual = NULL;
}

// Destrutor da classe NodoListaAtividade
NodoListaAtividade::~NodoListaAtividade()
{
	cout << "Destruindo roteiro de producao" << endl;
	if (Header == NULL)
		cout << "Lista Vazia" << endl;
	else {
		while (Header->ProximaAtividade != NULL){
			Atual = Header;
			Header = Header->ProximaAtividade;
			free(Atual);
		}
		cout << "Roteiro vazio" << endl;
	}
}

bool NodoListaAtividade::insereAtividade(Atividade *ativ)
{
	NodoListaAtividade *Novo;

	// Aloca ponteiro
	Novo = (NodoListaAtividade*)malloc(sizeof(NodoListaAtividade));
	// Teste de alocacao
	if (Novo == NULL)
		return false;
	else {
		Novo->RefAtividade = ativ;
		Novo->ProximaAtividade = NULL;
		if (Header == NULL){
			Header = Novo;
			Atual = Novo;
		}
		else
			Fim->ProximaAtividade = Novo;
		Fim = Novo;
	}
	return true;
}

// Se for a ultima atividade, retorna verdadeiro
// Caso contrario, retona falso
bool NodoListaAtividade::ultimaAtividade()
{
	if (Atual->ProximaAtividade == NULL)
		return true;
	else return false;
}
```

```
//Seleciona proxima atividade
Atividade* NodoListaAtividade::proximaAtividade(bool primAtiv)
{
      if (!primAtiv)
            Atual = Atual->ProximaAtividade;

      return Atual->RefAtividade;
}

void NodoListaAtividade::voltaInicioLista()
{
      Atual = Header;
}
```

Classe `Peca`

- Peca.h

```
#ifndef PECA_H
#define PECA_H

#include "NodoListaAtividade.h"

class Peca {
public:
      Peca(int);              // Construtor
      virtual ~Peca();  // Destrutor
      int getCodpeca();
      bool iniciaProducao(int);
      void criaRoteiro(NodoListaAtividade*);

private:
      int Cod_peca;
      NodoListaAtividade *refRoteiro;
};

#endif
```

- Peca.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Peca.h"
#include "Atividade.h"
#include "NodoListaAtividade.h"

// Construtor da classe Peca
Peca::Peca(int cod)
{
      Cod_peca = cod;
}

// Destrutor
Peca::~Peca()
```

```
{     }

// Retorna Cod_peca
int Peca::getCodpeca()
{
      return Cod_peca;
}

// Metodo que dispara a producao das pecas
bool Peca::iniciaProducao(int quantidade)
{
      bool primAtiv = true;
      bool ativCompleta = true;
      int pecaAtual = 1;
      Atividade *atividadeAtual;

      while (pecaAtual <= quantidade){
            cout << "****************** Iniciando a producao da peca ";
            cout << pecaAtual;
            cout << " ******************" << endl;
            // Seleciona proxima atividada
            while (!refRoteiro->ultimaAtividade()){
                if (primAtiv){
                     atividadeAtual=refRoteiro-proximaAtividade(true);
                     primAtiv = false;
                }
                elseatividadeAtual=refRoteiro->proximaAtividade(false);
                // Linhas de controle
                cout << "Atividade: ";
                cout << atividadeAtual->getCod_Atividade();
                cout << "         Descricao: ";
                cout << atividadeAtual->getDescricao();
                cout << "         Tempo Padrao: ";
                cout << atividadeAtual->getTempo_padrao() << endl;
                // Executa a atividade
                if (ativCompleta)
                   ativCompleta            =            atividadeAtual-
>executaAtividade(refRoteiro->ultimaAtividade());
                   else break;
            }
            refRoteiro->voltaInicioLista();
            primAtiv = true;
            pecaAtual++;
      }
      return ativCompleta;
}

void Peca::criaRoteiro(NodoListaAtividade *roteiro)
{
      refRoteiro = roteiro;
}
```

Classe `Pedido`

- Pedido.h

```
#ifndef PEDIDO_H
#define PEDIDO_H

#include "Peca.h"

class Pedido {
public:
      Pedido();               // Construtor
      virtual ~Pedido();      // Destrutor
      int getDia();
      int getMes();
      int getAno();
      int getQuantpecas();
      int getCodpeca();
      bool novoPedido(int, int, int, int, int);
      void criaRelacionamento(Peca*);

private:
      int Data_dia;
      int Data_mes;
      int Data_ano;
      int Quant_pecas;
      int Cod_peca;
      Peca *refPeca;
};

#endif
```

- Pedido.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Pedido.h"
#include "Peca.h"

// Construtor da classe Pedido
Pedido::Pedido()
{
      Data_dia = Data_mes = Data_ano = Quant_pecas = Cod_peca = 0;
}

Pedido::~Pedido()
{     }

int Pedido::getDia()
{
      return Data_dia;
}

int Pedido::getMes()
{
      return Data_mes;
}
```

```
int Pedido::getAno()
{
	return Data_ano;
}

int Pedido::getQuantpecas()
{
	return Quant_pecas;
}

int Pedido::getCodpeca()
{
	return Cod_peca;
}

bool Pedido::novoPedido(int dia, int mes, int ano, int quant, int cod)
{
	bool pedidoProduzido = true;

	cout << "Pedido referente a peca ";
	cout << refPeca->getCodpeca();
	cout << " incluido." << endl;

	pedidoProduzido = refPeca->iniciaProducao(quant);
	if (pedidoProduzido){
		cout << "Peca ";
		cout << refPeca->getCodpeca();
		cout << " produzida com secesso." << endl;
		return true;
	}
	else {
		cout << "Problemas na producao da peca  ";
		cout << refPeca->getCodpeca();
		cout << ". Peca nao produzida." << endl;
		return false;
	}
}

void Pedido::criaRelacionamento(Peca *pecaPedida)
{
	refPeca = pecaPedida;
}
```

Classe `Recurso`

- Recurso.h

```
#ifndef RECURSO_H
#define RECURSO_H

#include "Entidade_Simulada.h"

class Recurso : public Entidade_Simulada{
public:
	Recurso(int, int);                  // construtor
	virtual ~Recurso();                 // Destrutor
	int getCod_recurso();         // retorna Cod_recurso
	int getEstado();              // retorna Estado
```

```
      bool aloca_recurso(int);      //  Modifica  estado  para  ocupado,
recebe o tempo de alocacao

private:
      int Cod_recurso;  // Codigo do recurso
      int Estado;       // Estado do recurso: livre=0 ou ocupado=1
};

#endif
```

- Recurso.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using std::cout;
using std::endl;

#include "Recurso.h"
// Definicao do construtor
Recurso::Recurso(int cod,int est)
      :Entidade_Simulada()
{
      Cod_recurso = cod;
      Estado = est;
}

// Destrutor
Recurso::~Recurso()
{     }

// Retorna Cod_recurso
int Recurso::getCod_recurso()
{
      return Cod_recurso;
}

// Retorna Estado
int Recurso::getEstado()
{
      return Estado;
}

// Definicao da funcao de alocacao do recurso
bool Recurso::aloca_recurso(int tempo)
{
      long t;
      int tempo_calc;

      srand(time(&t));
      if (Estado == 0)  // Se o recurso esta livre
      {
            Estado = 1;   // Estado ocupado
            cout << "Recurso alocado" << endl;
            tempo_calc = (tempo - (int)(tempo * 0.1));
            tempo = tempo_calc + (rand() % (int)(tempo * 0.2));
            cout << "Tempo de execucao da atividade: ";
            cout << tempo << endl;
            this->simula(tempo);
```

```
            Estado = 0;
            cout << "Recurso liberado" << endl;

            return true;
      }
      else {
            cout << "Recurso Ocupado" << endl;
            return false;
      }
}
```

Classe `Relogio`

- Relogio.h

```
#ifndef RELOGIO_H
#define RELOGIO_H

class Relogio {
public:
      Relogio();              // Construtor
      virtual ~Relogio();     // Destrutor
      int getTempo();
      void incrementaRelogio(int);

private:
      int Tempo;
};

#endif
```

- Relogio.cpp

```
#include <iostream>

using std::cout;
using std::endl;

#include "Relogio.h"

Relogio::Relogio()      // Defincao do construtor
{
      Tempo = 0;
}

Relogio::~Relogio()
{     }

int Relogio::getTempo()
{
      return Tempo;
}

void Relogio::incrementaRelogio(int time)
{
      Tempo = Tempo + time;
}
```

Classe Torno

- Torno.h

```
# include "Torno.h"

// Construtor para a classe Torno
Torno::Torno(int cod, int est, int prep, char t[30], int mag)
      :Equipamento (cod, est, prep, t)
{
      Magazine = mag;
}

// Destrutor
Torno::~Torno()
{     }

// Retorna a quantidade de ferramentas no magazine
int Torno::getMagazine()
{
      return Magazine;
}
```

- Torno.cpp

```
# include "Torno.h"

// Construtor para a classe Torno
Torno::Torno(int cod, int est, int prep, char t[30], int mag)
      :Equipamento (cod, est, prep, t)
{
      Magazine = mag;
}

// Destrutor
Torno::~Torno()
{     }

// Retorna a quantidade de ferramentas no magazine
int Torno::getMagazine()
{
      return Magazine;
}
```

Classe Transporte

- Transporte.h

```
#ifndef TRANSPORTE_H
#define TRANSPORTE_H

#include "Recurso.h"

// Classe derivada da classe Recurso
class Transporte : public Recurso {
public:
      Transporte(int, int, int, int, char*);    // Construtor
      virtual ~Transporte();                          // Destrutor
```

```
      int getVelocidade();
      int getCarga();
      char* getTipo();

private:
      int Velocidade;
      int Carga_maxima;
      char Tipo[30];
};

#endif
```

- Transporte.cpp

```
#include <string>

#include "Transporte.h"

// Construtor para a classe Transporte
Transporte::Transporte(int cod, int est, int vel, int car, char t[30])
      :Recurso(cod, est)
{
      Velocidade = vel;
      Carga_maxima = car;
      strcpy(Tipo, t);
}

// Destrutor
Transporte::~Transporte()
{     }

int Transporte::getVelocidade()
{
      return Velocidade;
}

int Transporte::getCarga()
{
      return Carga_maxima;
}

char* Transporte::getTipo()
{
      return Tipo;
}
```

Função `Main`

```
#include <iostream>

using std::cout;
using std::endl;

#include "Pedido.h"
#include "Atividade.h"
#include "Torno.h"
#include "Furadeira.h"
#include "Area_Armazenamento.h"
#include "Fresa.h"
#include "Laminadora.h"
```

```
#include "Peca.h"
#include "NodoListaAtividade.h"
#include "Estrutura_Transporte.h"
#include "Relogio.h"

#define LIVRE           0
#define OCUPADO         1

int main ()
{
      bool fimProducao = true;

      // Primeira fase - criacao das intancias da simulacao
      Pedido pedido;
      //peca(cod_peca);
      Peca peca(15228);
      Relogio clock;

      //atividade(cod_atividade, Descricao, Tempo_Padrao);
      Atividade armazenarE(2, "Armazenar Entrada", 30);
      Atividade centrar(10, "Centrar", 300);
      Atividade tornear(15, "Tornear", 600);
      Atividade torn_final(20, "Torneamento Final", 900);
      Atividade fur_multiplo(25, "Furacao multipla", 480);
      Atividade furar(30, "Furacao da flange", 540);
      Atividade dentear(35, "Criar os dentes", 480);
      Atividade laminar(40, "Laminar a rosca", 90);
      Atividade chavetar(45, "Fresar chaveta", 180);
      Atividade armazenarS(50, "Armazenar Saida", 30);

      //Equipamento(cod_recurso, estado, tempo_preparacao, Tipo, ....)
      Area_Armazenamento Entrada(10,LIVRE,0,"Fila de Entrada",100, 0);
      Torno Centradeira(225, LIVRE, 4500, "Torno Centradeira", 5);
      Torno Heycomat(34, LIVRE, 5400, "Torno Heycomat Aut. 3/750", 2);
      Torno Daewoo(474,LIVRE,7200,"Torno Daewoo CNC Puma 350B", 5);
      Furadeira Brevet(97,LIVRE,4500,"Furadeira Multipla Brevet",300);
      Furadeira Clever(469,LIVRE,3000,"Furadeira Radial Clever", 500);
      Fresa Zewz(83, LIVRE, 3600, "Fresa Caracol Mor. Zewz", 5);
      Laminadora Rm(107, LIVRE, 2700, "Laminadora de Rosca RM", 6);
      Fresa Wmw(68, LIVRE, 4500, "Fresa Chaveteira Wmw", 2);
      Area_Armazenamento Saida(20, LIVRE, 0, "Fila de Saida", 100, 0);

      Estrutura_Transporte Agv1(15);
      Estrutura_Transporte Agv2(15);
      Estrutura_Transporte Agv3(15);
      Estrutura_Transporte Agv4(15);

      NodoListaAtividade roteiroProducao;

      // Segunda fase - criacao do roteiro de producao
      peca.criaRoteiro(&roteiroProducao);
      roteiroProducao.insereAtividade(&armazenarE);
      roteiroProducao.insereAtividade(&centrar);
      roteiroProducao.insereAtividade(&tornear);
      roteiroProducao.insereAtividade(&torn_final);
      roteiroProducao.insereAtividade(&fur_multiplo);
      roteiroProducao.insereAtividade(&furar);
      roteiroProducao.insereAtividade(&dentear);
      roteiroProducao.insereAtividade(&laminar);
      roteiroProducao.insereAtividade(&chavetar);
      roteiroProducao.insereAtividade(&armazenarS);
```

```
      // Terceira fase - criacao dos relacionamentos entre as
atividades e as maquinas
      pedido.criaRelacionamento(&peca);
      armazenarE.criaRelacionamentoEquip(&Entrada);
      centrar.criaRelacionamentoEquip(&Centradeira);
      tornear.criaRelacionamentoEquip(&Heycomat);
      torn_final.criaRelacionamentoEquip(&Daewoo);
      fur_multiplo.criaRelacionamentoEquip(&Brevet);
      furar.criaRelacionamentoEquip(&Clever);
      dentear.criaRelacionamentoEquip(&Zewz);
      laminar.criaRelacionamentoEquip(&Rm);
      chavetar.criaRelacionamentoEquip(&Wmw);
      armazenarS.criaRelacionamentoEquip(&Saida);
      armazenarE.criaRelacionamentoTransp(&Agv1);
      centrar.criaRelacionamentoTransp(&Agv1);
      tornear.criaRelacionamentoTransp(&Agv1);
      torn_final.criaRelacionamentoTransp(&Agv2);
      fur_multiplo.criaRelacionamentoTransp(&Agv2);
      furar.criaRelacionamentoTransp(&Agv3);
      laminar.criaRelacionamentoTransp(&Agv4);
      dentear.criaRelacionamentoTransp(&Agv4);
      chavetar.criaRelacionamentoTransp(&Agv4);

      // Quarta fase - inicio da producao
      fimProducao = pedido.novoPedido(20, 10, 2002, 30, 15228);
      cout << "*****************************************" << endl;
      cout << "     Tempo Total de Producao: ";
      cout << clock.getTempo() << endl;
      cout << "*****************************************" << endl;
      if (fimProducao)
            cout << "Producao finalizada com sucesso" << endl;
      else cout<<"Producao finalizada com problemas no pedido"<< endl;

      return 0;
}
```

Bibliografia

AUTO SIMULATIONS, Inc. **Automod User's Manual**. Bountiful, 1999. v.1.

AUTO SIMULATIONS, Inc. **Automod User's Manual**. Bountiful, 1999. v.2.

AUTO SIMULATIONS, Inc. **Drawing ACE Graphics Tutorial**. Bountiful, 1999.

AUTO SIMULATIONS, Inc. **Automod Version 9.0**. Bountiful, 2002. 1 CD-ROM.

BALDUZZI, Fabio; BRUGALI, Davide. Decentralised Control Strategies Via Agent Interactions. In: IFAC WORKSHOP ON MULTI-AGENT SYSTEMS IN PRODUCTION, MAS, 1., 1999, Vienna, Austria. **Preprints**. [S.l.: IFAC], 1999. p.133-138.

BANKS, Jerry; CARSON, John S. II; NELSON, Barry L. **Discrete-Event System Simulation**. 2nd ed. New Jersey: Prentice-Hall, 1995.

BATHAGLINI, Mairon. **Um Estudo sobre Orientação a Objetos e Simulação Interativa Visual Aplicados a Sistemas de Manufatura**. 1997. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECKER, Leandro Buss. **Ambiente de Modelagem e Implementação de Sistemas de Tempo Real Usando o Paradigma de Orientação a Objetos**. 1999. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BERTOTTO, Carlos Alberto. **Estudo sobre Metodologias de Modelagem, Controle e Simulação em Sistemas de Manufatura**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BERTOTTO, Carlos Alberto; PEREIRA, Carlos Eduardo.; HENRIQUES, Renato Ventura Bayan. Object-Oriented Simulation and Manufacturing Systems Simulation Environments: Case Study. In: EUROPEAN SIMULATION MULTICONFERENCE, ESM, 15., 2001, Prague, Czeck Republic. **Proceedings…** San Diego: SCS, 2001. p. 151-153.

BOOCH, Grady. **Object Oriented Analysis and Design**. New York: The Benjamin/Cummings, 1994.

BOOCH, Grady. **UML**: Guia do Usuário. Rio de Janeiro: Campus, 2000.

COPSTEIN, Bernardo. **SIMOO**: Plataforma Orientada a Objetos para Simulação Discreta Multi-Paradigma. 1997. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DEITEL, Harvey; DEITEL, Paul. **C++ Como Programar**. 3.ed. Porto Alegre: Bookman, 2001.

DOWNING, Douglas; CLARK, Jeffrey. **Estatística Aplicada**. São Paulo: Saraiva, 1998.

FOWLER, Martin; SCOTT, Kendall. **UML Essencial**. 2.ed. Porto Alegre: Bookman, 2000.

GAMMA, Erich et al. **Padrões de Projeto**. Porto Alegre: Bookman, 2000.

GOVINDARAJ, T. et al. Manufacturing Simulation Using Objects. In: SUMMER COMPUTER SIMULATION CONFERENCE, SCSC, 22., 1990, Alberta. **Proceedings…** San Diego: SCS, 1990. p.219-224.

HOLONIC MANUFACTURING SYSTEMS. **Holonic Manufacturing Systems**. 2002. Disponível em: <http://hms.ifw.uni-hannover.de/> Acesso em: 15 jul. 2002.

KOESTLER, Arthur. **The Ghost in the Machine**. London: Hutchinson & Co, 1967.

LANGER, Gilad; SORENSEN, Christian. Developing a System Architecture for Holonic Shop Floor Control. In: IFAC WORKSHOP ON INTELIGENT MANUFACTURING SYSTEMS, IMS, 5., 1998, Gramado, Brasil. **Preprints.** [S.l.: IFAC], 1998. p.57-63.

LAW, Averill. M.; KELTON, W. David. **Simulation Modeling and Analysis**. 2nd ed. New York: McGraw-Hill, 1991.

LIM, Ming Kim; ZHANG, Zhengwen. APPSS – An Agent-Based Dynamic Process Planning and Scheduling System. In: IFAC WORKSHOP ON MULTI-AGENT SYSTEMS IN PRODUCTION, MAS, 1., 1999, Vienna, Austria. **Preprints.** [S.l.: IFAC], 1999. p. 1-6.

MARIANI, Ivandro. **Apresentação da Célula de Ponteiras**. Caxias do Sul: Pigozzi S/A, 2000. Apresentação em Slides.

MARIANI, Ivandro. **Coleta de Dados sobre a Célula de Ponteiras**. Caxias do Sul, 25 nov. 2001. Entrevista concedida a: Carlos Alberto Bertotto.

MARÍK, Vladimír et al. Application of the Multi-Agent Approach in Production Planning and Modeling. In: IFAC WORKSHOP ON INTELIGENT MANUFACTURING SYSTEMS, IMS, 5., 1998, Gramado, Brasil. **Preprints.** [S.l.: IFAC], 1998. p. 131-136.

MICROSOFT CORPORATION. **Microsoft Visual C++ 6. 0 Introductory Edition**. Porto Alegre, 1998. 1 CD-ROM.

MOWBRAY, Thomas; RUH, William. CORBA Basics. In: MOWBRAY, Thomas; RUH, William. **Inside CORBA**. Reading: Addison Wesley, 1997. cap. 1, p. 3-18.

MSE MANUFACTURING STRATEGIES LTDA. **Sinopse de alguns Projetos**. São Paulo, 1999. 1 CD-ROM. Word for Windows.

NARAYANAN, S. et al. Object-Oriented Simulation to Support Modeling and Control of Automated Manufacturing Systems. In: WESTERN MULTI CONFERENCE, WMC, 1., 1992, San Diego, **Proceedings…** [S.l.: s.n.], 1992. p. 59-63.

NARAYANAN, S. et al. Research in Object-Oriented Manufacturing Simulations: An Assessment of the State of the Art. **IIE Transactions**, [S.l.], v. 30, n. 9, p. 795-810, Sept. 1998.

OBJECT MANAGEMENT GROUP. **CORBAManufacturing Domain**. 2002. Disponível em: <http://www.omg.org/homepages/mfg> Acesso em: 7 jun. 2002.

ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. ORB Fundamentals. In: ORFALI, Robert; HARKEY, Dan; EDWARDS, Jeri. **Instant CORBA**. New York: John Wiley & Sons, 1997. cap 2, p. 51-102.

PARK, Tea Y.; HAN, Kwan H.; CHOI, Byoung K. An Object-Oriented Modelling Framework for Automated Manufacturing System. **Computer Integrated Manufacturing**, [S.l.], v. 10, n. 5, p. 324-334, May 1997.

PEREIRA, Carlos Eduardo. Real Time Active Objects in C++/Real-Time UNIX. In: WORKSHOP ON LANGUAGES, COMPILER AND TOOL SUPPORT FOR REAL-TIME SYSTEMS, 1994, Orlando. **Proceedings...** [S.l.:ACM], 1994. 6 p.

PEREIRA, Carlos Eduardo et al. Applying Multi-Agent Systems to Real-Time Industrial Automation. In: IFAC WORKSHOP ON MULTI-AGENT-SYSTEMS IN PRODUCTION, MAS, 1., 1999, Vienna, Austria. **Preprints**. [S.l.: IFAC], 1999. p. 127-131.

RATIONAL SOFTWARE CORPORATION. **Rational Rose 98**. 1998. 1 CD-ROM.

ROBERTS, Chell A.; DESSOUKY, Yasser M. An Overview of Object-Oriented Simulation. **Simulation**, [S.l.], v. 70, n. 6, p.359-368, June 1998.

ROSINHA, L. F. F. et al. Arquitetura de Simulação Flexível com Animação Gráfica Aplicada ao Projeto de FMS. In: CONGRESSO BRASILEIRO DE AUTOMÁTICA, CBA, 13., 2000, Florianópolis, Brasil. **Anais...** Florianópolis: SBA, 2000. p. 1012-1017.

RUMBAUGH, J. et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Campus, 1994.

SANZ-BOBI, Miguel A. et al. Multi-Agent Diagnosis Systems in Industry. In: IFAC WORKSHOP ON MULTI-AGENT SYSTEMS IN PRODUCTION, MAS, 1., 1999, Vienna, Austria. **Preprints**. [S.l.: IFAC], 1999. p. 259-264.

SILVA, Nuno; SOUZA, Paulo; RAMOS, Carlos. Proposal for a Dynamic Scheduling Architecture and Algorithm using an Holonic Approach. In: IFAC WORKSHOP ON INTELLIGENT MANFACTURING SYSTEMS, IMS, 5., 1998, Gramado, Brasil. **Preprints**.[S.l.: IFAC], 1998. p. 71-74.

TANENBAUM, Andrew S. **Redes de Computadores**. 5.ed. Rio de Janeiro: Campus, 1997.