Álvaro Francisco de Castro Medeiros

Extensão à STL para Representação Canônica de Conceitos em Métodos de Especificação de Software Orientadas a Objetos

Tese submetida à Coordenação dos Cursos de Pós-Graduação em Engenharia Elétrica da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências em Engenharia Elétrica.

Área de Concentração: Processamento da Informação

Orientadores: Prof. José Antão Beltrão Moura[1] – PhD
Paul Jorgensen[2] – PhD.

Campina Grande, Paraíba, Brasil
2001

[1] Universidade Federal da Paraíba, Campina Grande, Paraíba.
[2] Grand Valley State University, Allendale, Michigan, USA.

M488e Medeiros, Alvaro Francisco de Castro
Extensao a STL para representacao canonica de conceitos em metodos de especificacao de software orientadas a objetos / Alvaro Francisco de Castro Medeiros. - Campina Grande, 2001.
210 f.

Tese (Doutorado em Engenharia Eletrica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Comunicacao de Dados 2. Ferramentas CASE 3. Metodos de Especificacao de Software Orientados a Objetos 4. Tese - Engenharia Eletrica I. Moura, Jose Antao Beltrao II. Jorgensen, Paul III. Universidade Federal da Paraiba - Campina Grande (PB) IV. Título

CDU 681.327.8(043)

EXTENSÃO À STL PARA REPRESENTAÇÃO CANÔNICA DE CONCEITOS EM METODOLOGIAS DE ESPECIFICAÇÃO DE SOFTWARE ORIENTADAS A OBJETOS

ÁLVARO FRANCISCO DE CASTRO MEDEIROS

Tese Aprovada em 08.05.2001

PROF. JOSÉ ANTÃO BELTRÃO MOURA, Ph.D., UFPB
Orientador

PROF. PAUL C. JORGENSEN, Ph.D., Grand Valley State University
Co-Orientador

PROF. ALEXANDRE MARCOS LINS DE VASCONCELOS, Dr., UFPE
Componente da Banca

JOAQUIM CELESTINO JÚNIOR, Dr., UFC
Componente da Banca

FRANCISCO VILAR BRASILEIRO, Dr., UFPB
Componente da Banca

ANGELO PERKUSICH, D.Sc., UFPB
Componente da Banca

ULRICH SCHIEL,Dr.rer.nat., UFPB
Componente da Banca

CAMPINA GRANDE - PB
Maio - 2001

Agradecimentos

- Aos Professores Ulrich Schiel e Alexandre Lins pela correção e comentários que direcionaram o trabalho na reta final;
- Ao Prof. José Antão Beltrão Moura pela dedicação e constante paciência na busca das soluções dos problemas e pela orientação em toda a extensão deste trabalho;
- Ao Prof. Paul Jorgensen pela orientação e apoio em grande parte deste trabalho. Agradeço a oportunidade de utilização do espaço e recursos que me foram disponibilizados na Grand Valley State University (GVSU– Michigan- EUA);
- À Profª Islene Mangueira Soares pela correção ortográfica;
- Aos Professores Hélio Menezes, José Antônio, e Antonio Carlos pelo incentivo, dicas, comentários e sugestões que muito ajudaram na elaboração deste trabalho;
- Ao Prof. Dr. Antônio Marcos Nogueira, coordenador de pós-graduação em Engenharia Elétrica (COPELE) da UFPB, pelo apoio demonstrado;
- À CAPES/PICD e ao CNPq pelo apoio financeiro, através de bolsa de estudos e suporte para visitas à GVSU;
- À Profª Maria Elizabeth do Departamento de Ciências da Computação da UFAL pelo companheirismo e incentivo no início deste trabalho.
- À Ângela e Pedro da COPELE meu muito obrigado.

Dedico este trabalho à minha esposa Isménia e às nossas filhas Anna Carolina e Amanda pelo apoio e compreensão nas inúmeras horas em que estive ausente do convívio com elas para poder elaborar esta tese.

"Nós somos o que fazemos. Sobretudo o que fazemos para modificar o que somos"

Eduardo Galeano

Resumo

Esta pesquisa propõe extensões à Linguagem de Transferência de Semântica (*Semantic Transfer Language-STL*) — parte do Padrão IEEE 1175 para comunicação entre ferramentas CASE — para que esta possa transferir informações entre métodos distintos de especificação de software orientados a objetos (MESOOs).

Foi introduzido o conceito de *Equivalência* (*Eq*) através de um novo metamodelo para STL que será usado para identificar equivalência entre os MESOOs selecionados para transferência. Desenvolvemos um interpretador de pacotes que, utilizando o *Eq*, poderá descobrir conceitos equivalentes ao longo da transmissão de pacotes STL permitindo reaproveitamento de Especificações legadas.

Abstract

This research proposes an extension to the Semantic Transfer Language (STL) — part of the CASE tools interconnection IEEE 1175 Standard — to represent object-oriented software specification methodologies (OOSMs).

We introduced the *Equivalence* Concept (*Eq*) in the STL Language through a new metamodel and use it to represent equivalence between the OOSMs selected for this research. Using *Eq* it will be possible to re-use OOSM legacy specification.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo

1

Introdução

Vivemos na era da globalização. Computadores estão ligados em uma rede mundial possibilitando o desenvolvimento de produtos de forma distribuída e cooperativa. Para serem competitivas no mercado, companhias de sucesso estão procurando cada vez mais a qualidade e a produtividade, através da utilização de ambientes padronizados e automatizados, [MEDE 94, 98]. Apesar disto, a crise do software continua [SOMM97, PRESS97]. A indústria de hardware disponibiliza produtos cujos recursos demoram a ser explorados devido à dificuldade encontrada para desenvolver sistemas de software complexos para atender às necessidades dos usuários cada vez mais exigentes [MOOR 99, JACO 99]. Um esforço comum da indústria do software, para acompanhar a evolução do hardware, pode-se ver através do número crescente de notações ou métodos para especificação de software orientadas a objetos (MESOOs[3]) atualmente disponíveis [BHAR 99].

[3] Vamos utilizar o termo método para referenciar uma notação ou representação gráfica com o seu respectivo significado.

O uso de técnicas e MESOOs que visem a automatização do processo de desenvolvimento de software apresenta-se como uma resposta à crise do software, no qual a produção de programas não acompanha a mesma velocidade de evolução da indústria do hardware [SOMM 97].

Os MESOOs utilizam modelos para representar o domínio do problema e muitas vezes contam com o auxílio do próprio computador para agilizar o processo de desenvolvimento de software. Podemos encontrar empresas usando ferramentas CASE[4], que implementam ou automatizam um ou mais MESOOs, para ajudar na construção de software [BERR 98]. Entretanto, esta agilização do processo não possibilita o reaproveitamento ou reutilização de componentes ou artefatos anteriormente produzidos, pois nem sempre é possível a transferência de semântica entre ferramentas CASE produzidas por diferentes fornecedores, fazendo com que as especificações já existentes (ou parte delas) sejam reaproveitadas.

Podemos ilustrar os benefícios da comunicação entre ferramentas CASE através do reaproveitamento de artefatos de software produzidos por um MESOO já obsoleto mas que pode ser reaproveitado numa nova perspectiva de visão canônica. A necessidade desta padronização pode ser ainda visualizada através de situações reais operacionais nas quais haja limitações de recursos ou existam preferências pessoais, relacionadas com a usabilidade, eficiência ou funcionalidade de duas ou mais ferramentas que suportem $M = \{m_1, m_2, m_3 \text{ e } m_4\}$ que são MESOOs usados para especificar uma aplicação A. Neste ambiente, podem ocorrer diversas

4 Sigla do idioma Ingles ("*Computer Aided Software Engineering*").

situações relacionadas às restrições associadas à política de licenças de uso dos softwares FC_{aA} e FC_{bA} (duas ferramentas CASE a e b usadas para especificar A através de $m \in M$).

Uma mesma ferramenta, digamos FC_{aB} pode suportar os mesmos MESOOs que FC_{bA} mas, por possuir uma melhor interface, tem a preferência dos usuários. Já FC_{bA} é reconhecidamente a ferramenta que melhor gera código. Neste caso, pode-se usar FC_{aA} e FC_{bA} de forma cooperativa. Uma faz a especificação e transfere informação para que a outra possa gerar o código.

Uma situação mais próxima da realidade pode ser visualizada em ambiente onde exista heterogeneidade de hardware. Geralmente, as máquinas mais robustas têm o preço da licença de uso do software mais caro. Digamos que FC_{aA} funcione em uma plataforma UNIX® em uma máquina servidora. As máquinas cliente utilizam FC_{bA} no ambiente operacional Windows®. Não é absurdo imaginar que FC_{aA} é muitas vezes mais cara que FC_{bA}. Neste contexto, a necessidade de comunicação entre as ferramentas é imperativa. Através dela, os custos poderão ser minimizados ou a produtividade aumentada a partir da seleção das ferramentas corretas para o perfil de cada usuário. Obviamente, esta comunicação só será possível se o fabricante da ferramenta quiser implementa-la.

1.1 Comunicação entre Ferramentas CASE

Com a disseminação da Internet e a distribuição das aplicações, os usuários tornam-se cada vez mais exigentes. Atividades de uso de ferramentas isoladas passam por uma mudança de paradigma onde a comunicação é necessária. A arquitetura cliente-

servidor ganha espaço como nova abordagem para descentralizar e distribuir informações. Neste contexto, a comunicação entre ferramentas distintas passa a ter um papel fundamental. Não apenas para transmitir informações entre ferramentas CASE, mas também para possibilitar a comunicação de forma mais produtiva entre aplicações como *browserr* de hipertextos, sem a intervenção do usuário. Como podemos ver em Reed Hellman [HELL 99], é necessário o estabelecimento de linguagens com maior poder de transferência de semântica. Mas, esta condição não é suficiente. É preciso que padrões sejam adotados para que as ferramentas possam se comunicar corretamente.

Podemos citar o exemplo concreto dos hipertextos da Web. Neste caso, os clientes e servidores precisaram utilizar uma linguagem de transferência de semântica mais poderosa que permitisse a comunicação mais eficiente sem a intervenção dos usuários. Um cliente quer saber qual o "preço" mis em conta de um produto, por exemplo. É preciso que todos os servidores também conheçam "preço" para que a comunicação seja feita de forma correta. Reuven M. Lerner [LERN 2000] ilustra a necessidade de estabelecimento de um padrão na construção dos rótulos ou padrões que a comunidade deve adotar para que a transferência de semântica ocorra.

A transferência de semântica é fundamental ao longo do processo de desenvolvimento de sistemas onde ferramentas CASE são utilizadas para representar e automatizar as diversas fases do desenvolvimento do software. Como podemos ver no mercado, as ferramentas que suportam os MESOOs apresentam uma arquitetura geral composta de um editor gráfico, um dicionário ou banco de

dados para armazenar as características do modelo de representação da realidade, um módulo que faz a checagem semântica e, às vezes, geradores de código fonte para uma linguagem de programação específica.

O problema com a transferência de semântica é que as ferramentas CASE utilizam um formato proprietário de armazenamento para representar a especificação de uma aplicação A e não possibilitam o compartilhamento direto dos dados por outras ferramentas. Por exemplo, suponhamos que existam duas ferramentas CASE FC_{aA} e FC_{bA} que implementem conceitos $C = \{c_1, c_2, c_3, \}$ de um MESOO $m \in M$, através de representações gráficas estabelecidas em diagramas $\{d\}$, como ilustra a Figura 1. No âmbito desta tese, M é o conjunto dos MESOOs selecionados

$$M = \{AOO, SHAM, UML, WOOD\}$$

e m pode assumir, respectivamente, cada um dos métodos de especificação de software objeto de estudo desta tese ([COAD 90, 91], [SHLA 88, 91] [UML 96, 99] e [WIRF 90]). O problema é que FC_a e FC_b utilizam *layouts* internos proprietários para a representação dos conceitos C presentes nos diagramas de M. Não há, portanto, uma preocupação por parte dos desenvolvedores de ferramentas CASE em estabelecer mecanismos para a transferência de semântica.

Como podemos observar, a importância da transferência de semântica está relacionada com a possibilidade de construção de ambientes que integrem diferentes ferramentas e proporcione uma maior reutilização de artefatos de software produzidos ao longo de seu processo de construção nas fases anteriores à codificação.

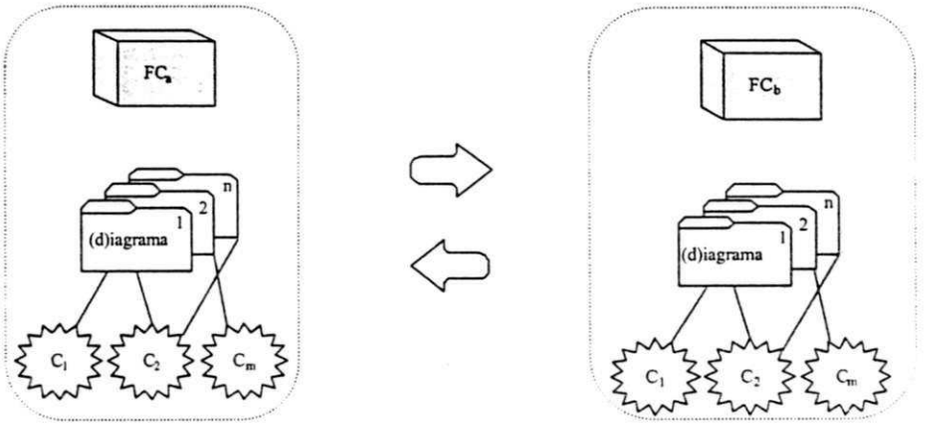

Figura 1: Necessidade de Transferência de Semântica entre *FC,*

A transferência de semântica vem sendo discutida sob diversos aspectos na literatura. Do ponto de vista dos MESOOs, há um esforço no sentido de se estabelecer uma notação que reúna as funcionalidades presentes nos principais métodos disponíveis. Podemos citar o Método *Fusion* [COLE 94]. A proposta de Coleman tentou fundir as melhores características dos MESOOs em um único método de especificação de software orientado a objetos, na tentativa de criar um padrão. Na realidade, o que ocorreu de fato foi o surgimento de mais um MESOO. Nota-se, portanto, que qualquer iniciativa de unificação tem que contar com a participação do mercado (desenvolvedores de ferramentas CASE e usuários dos MESOOs) para que tenhamos, em longo prazo, o estabelecimento de padrões de fato. Nesta direção, podemos apontar o exemplo da comunidade que trabalha com o objetivo de estabelecer UML [UML 96-2000] como um padrão para a indústria.

A idéia central em UML é semelhante à de Coleman *et al* [COLE 94] de criar uma única forma unificada de representar os

conceitos presentes nos principais MESOOs disponíveis. A abordagem aqui foi diferente, pois houve um apelo muito forte para que o mercado pudesse participar de forma decisiva na evolução do modelo. O fato desta iniciativa ter sido encabeçada por James Rumbaugh, Grady Booch e Ivar Jacobson, líderes de três dos mais aceitos métodos de especificação de software orientados a objetos (OMT [RUMB 91], Booch [BOOC 91] e OOSE [JACO 92], respectivamente) deu uma maior visibilidade e publicidade para que desenvolvedores e consumidores passassem a considerar UML. Mesmo que UML venha a tornar-se um padrão para a indústria temos que considerar a possibilidade da existência de um legado considerado em outros MESOOs que justifique o esforço para implementação de um mecanismo de conversão entre estes.

A reutilização de artefatos de software desde o início de seu processo de produção é um desejo das comunidades acadêmica e industrial para fazer frente à crise do software [SOMM 97]. Apesar das diferenças culturais existentes entre a academia e a indústria [ZEIG 2000], a literatura tem apresentado sugestões que suportam a construção de ambientes de desenvolvimento cada vez mais integrados. Nas iniciativas de Roberto Bellizona *et al* [BELL 95], Hans W. Nissen [NISS 96] e Alan Davis *et al* [DAVI 97] temos três ambientes com definição de metamodelos que permitem a integração de diferentes perspectivas em uma mesma ferramenta CASE. Entretanto, não há como transferir semântica entre elas. Estas iniciativas porém, do ponto de vista de representação de diagramas dos MESOOs, essencialmente propõem três novos MESOOs

(Método ITHACA, PFR[5] e CR, respectivamente). Uma verificação mais apurada das ferramentas CASE RECAST [BELL 95], USU[6] [NISS 96] e TINA [DAVI 97] apresenta a mesma limitação - falta de compartilhamento (ou de transferência) de semântica de informações, digamos de diagramas de um mesmo MESOO m, entre ferramentas CASE diferentes FC_{aA} e FC_{bA} para representar A, devido às suas estruturas proprietárias de armazenamento.

Outro problema é a necessidade de conversão de especificações existentes em um MESOO, ou parte dela, para outro mais atual ou que tenha determinadas características que justifique o reaproveitamento.

Observe que o problema de comunicação tem duas ordens de magnitude: a primeira operacional para fazer com que os pacotes de informações sejam transferidos entre duas ferramentas CASE; e o segundo, possibilitar que os conceitos destes pacotes sejam tratados ao longo de seu processo de mapeamento ou conversão – fazendo com que considerações entre métodos sejam endereçadas.

Uma solução para este problema é a adoção de uma forma genérica de comunicação entre ferramentas CASE distintas, preservando todo o investimento já realizado, através de um padrão adotado pelos desenvolvedores.

1.1.1 Escolha de um Padrão de Comunicação

Desde o início da década de noventa, quando os MESOOs começaram a ter as suas primeiras ferramentas CASE que automatizavam e suportavam o processo de criação de especificações,

[5] Sigla do idioma Ingles (*"Analysis of Presence and Future Requirements"*).

com geração de código inclusive, pairava a pergunta sobre quantos formatos padrões de troca de informações eram necessários para operacionalizar a interoperabilidade entre ferramentas CASE. [MEYE 2001]

Propostas surgiram através de organizações do porte da EIA (Electronic Industries Alliance) [WIRT 2001], IEEE (Institute of Electrical and Electronics Engineering) [IEEE 92-99] e, mais recentemente, da comunidade de OMG (Object Management Group) que tem, atualmente, uma forte participação da indústria. Todas as abordagens levam a uma forma de associar as informações a serem transferidas com um rótulo que descreve ou detalha características relacionadas com o conteúdo sendo trocado entre as ferramentas envolvidas na comunicação, como pode ser visto na Figura 2, trata-se de uma forma de descrever o catálogo de um pacote (conteúdo) de informações [DAMM 2000].

O mecanismo de comunicação do Padrão IEEE 1175 (a forma de descrever o catálogo) tem um nível a menos de abstração do que a forma de fazer a mesma atividade usando a Família de Padrões CDIF (CASE Data Interchange Format) [WIRT 2001]. Neste último caso, é necessário se estabelecer assuntos (subjects) e a partir daí, detalhar a estrutura da informação a ser transferida.

[6] Implementado em função do Ambiente ConceptBase.

Figura 2: CDIF Arquivo / IEEE 1175 – STL Específico

A Família de Padrões CDIF e o Padrão IEEE 1175 tendem a convergir devido ao seu objetivo de estabelecer padrões para operaciolanizar a comunicação entre ferramentas CASE no âmbito de uma organização. A diferença é que o último Padrão está mais voltado para a comunidade acadêmica, não exigindo pagamento de taxa para poder participar.

Por outro lado, com evolução da Internet e a popularização dos browsers, surgem aplicações cada vez mais sofisticadas exigindo que soluções para troca de informações entre aplicações sejam elaboradas de modo a permitir que páginas da Web sejam criadas unindo tecnologia de hipertexto e de banco de dados. Surge a necessidade de se extender HTML (Hyper Text Markup Language), criando XML (Extended Markup Language) que é uma forma de criar documentos que possam estabelecer um significado (semântica) prévio para informações a serem trocadas ("*tags*") entre aplicações. [OBRI 2000]

XML introduz nas páginas de textos HTML uma forma de descrever DTD (Data Type Definition) que nada mais são do que os rótulos que descrevem o conteúdo a ser transferido. Estes, por sua

[7] Tipo de "etiqueta" usanda em linguagem de marcação HTML para construção de páginas na Web.

vez, encontram grande dificuldade na representação dos gráficos utilizados pelos MESOOs [DAMM 2000].

A OMG e o peso de empresas como IBM, Unisys, Rational e outras estão, atualmente, trabalhando em uma solução que visa a construção de catálogos com maior poder de descrição (formas de parametrização e maior facilidade de geração) de DTDs de documentos XML. A proposta de criação do Padrão XMI (XML Metadata Interchange) é uma indicação de que a indústria precisa urgentemente estabelecer um padrão de fato que viabilize a comunicação entre ferramentas CASE. Trabalhando de forma otimista, o problema da extensão necessária para descrever gráficos (dos diagramas dos MESOOs) ainda vai ser um problema. Uma alternativa pode ser a construção de dialetos de XML/XMI do tipo GLX (Graph eXchange Language) [SIM 2000]para representar e particionar os diagramas em unidades de interesse para manipulação pelas ferramentas CASE.

Outras empresas estão tentando firmar seus próprios padrões como é o caso, apenas para citar um exemplo, da Microsoft que tenta impor seu formato XIF (XML Interchange Format) [www.Microsoft.com/xif] que é uma tecnologia complementar para o gerenciamento da solução chamada de repositório integrado de metadados, que não deixa de ser uma solução proprietária.

As possibilidades de escolha sobre qual padrão estudar, na época do início desta pesquisa, restringia-se a STL e CDIF. Resolvemos escolher o primeiro pelas seguintes razões: a proposta de utilização de um mecanismo mais genérico de comunicação que não envolvesse necessariamente o uso de arquivos como meio de

armazenamento; a definição do metamodelo em um formato que poderia ser automatizado com poucas modificações gerando uma representação em BNF; uma maior abertura para a participação da comunidade acadêmica relacionada às decisões com impacto na evolução do Padrão IEEE 1175.

1.2 Linguagem para Transferência de Semântica

A Linguagem de Transferência de Semântica – STL[8] nasceu com o propósito de operacionalizar a comunicação entre ferramentas CASE distintas (FC_{aA} e FC_{bA}) produzidas por diferentes fornecedores para representar especificações de aplicações A. Pesquisadores da indústria e da academia, em um esforço conjunto, formaram uma Comissão do IEEE de número 1175 para estudar e propor um padrão de referência para interconexão de ferramentas CASE [IEEE 92-99] no qual a definição de STL foi apresentada.

STL é uma linguagem declarativa, assim como a IDL[9] do ambiente CORBA[10]. STL não possui comandos de controle de fluxo ou de laços de repetição.

IDL está intimamente ligada a uma arquitetura, como podemos ver na Figura 3. IDL oferece mecanismos que permitem a definição de uma interface para que os serviços implementados através de objetos sejam compartilhados entre aplicações distribuídas. IDL está preocupada em manter a interoperabilidade entre os objetos implementados em plataformas diferentes. IDL mapeia as interfaces

[8] Sigla que vem das inicias do inglês STL ("*Semantic Transfer Language*").
[9] Linguagem de definição de interface ("*Inteface Definition Language*").
[10] Sigla que vem das iniciais CORBA do inglês ("*Common Object Request Broker Architecture*").

para linguagens específicas de cada domínio particular mantendo a independência de plataforma. Ao contrário de IDL que foi projetada para implementar componentes distribuídos através de uma arquitetura que contempla protocolos de comunicação e representação de objetos em plataformas heterogêneas, STL está preocupada com a transferência de pacotes entre ferramentas CASE distintas.

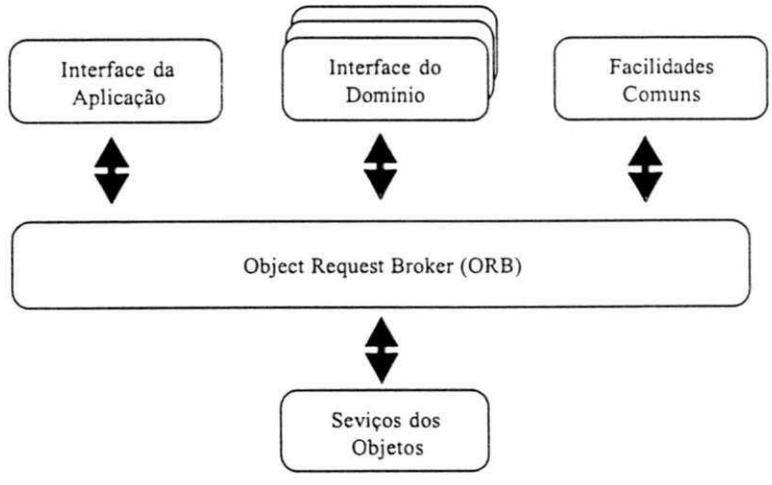

Figura 3: Linguagem IDL de CORBA

Em STL, a comunicação entre $FC_{a,i}$ e $FC_{b,i}$ é feita através de pacotes. Um pacote P é composto de um catálogo e uma descrição semântica (P = <*cat,desc*>). O Catálogo descreve o tipo e a quantidade dos conceitos $\{c\}$ presentes em um diagrama d parte de uma especificação $E_{A,m}$ escrita em um MESOO $m \,^{o} M$ que representa o domínio de um problema do mundo real de uma aplicação A. Podemos dizer que

$$E_{A,m} = \{d \mid d \in m \wedge d \text{ descreve aspectos de } A\}$$

$E_{A,m}$ é definido como o conjunto de todos os diagramas relevantes de m utilizados para delimitar um problema de interesse em A. Um

diagrama d pode representar um ou mais conceitos C ($C = \{c_1, c_2, \ldots, c_n\}$). Todo MESOO tem seu conjunto de conceitos. Cada um deles utiliza seus conceitos para descrever aplicações A.

O Catálogo STL permite a descrição de semântica e possibilita a transferência dos conceitos $\{c\}$, presentes em E_{Am}, através de Pacotes P de informações trocadas entre as diferentes ferramentas CASE FC_{aA} e FC_{bA}, $a \neq b$. Existem limitações no metamodelo STL, como por exemplo a falta de um metamodelo que descreva os conceitos dos MESOOs de uma forma a permitir um agrupamento de conceitos que possa ser referenciado ao longo do mapeamento em pacotes. Com a evolução dos MESOOs e das ferramentas CASE espera-se que os mecanismos de transferência de semântica sejam capazes de fazer algo mais do que operacionalizar a transmissão de pacotes P entre ferramentas. Com o surgimento de novos conceitos nos MESOOs, STL precisa aperfeiçoar seu metamodelo para que existam condições de registrar equivalência entre mapeamento dos conceitos dos MESOOs. A capacidade de combinação e a granularidade dos conceitos STL deve permitir representar duas especificações E_{Am} e E_{An} escritas em métodos m e n ($m \in M$, $n \in M$, $m \neq n$) e analisar, através de pacotes p a equivalência entre os conceitos dos MESOOs. Como discutiremos a seguir, o mapeamento entre conceitos é importante e pode representar ganhos de produtividade na produção de software.

Para ilustrar a limitação identificada no metamodelo atual de STL, podemos construir artefato de software *Istl.Atual*, para implementar o interpretador de pacotes e ativa-lo para interpretar pacotes, e verificar que não existe nenhuma informação entre dois

pacotes transferidos no tempo que possa ser usada pelas ferramentas CASE.

1.3 Usando STL para Transferência de Informações entre MESOOs

Em uma empresa as pessoas normalmente tornam-se especialistas em um MESOO m_1 qualquer e tendem a utilizá-lo para descrever especificações E_{Am1}, independente do tipo do domínio do problema (ou cenário) sendo tratado, mesmo existindo outro MESOO m_2 mais apropriado. Se devemos escolher m_1 ou m_2 como o método de especificação "correto" é uma questão difícil de responder e encontra resistências culturais. Muitas vezes, o fato de ser especialista no uso de um MESOO é que determina sua escolha. Por outro lado, alguns aspectos da realidade podem não ser expressos em E_{Am1} porque escolhemos a notação (MESOO) "errada".

Conhecer todos os MESOOs não vai nos poupar de arcar com a responsabilidade de escolher o método mais adequado para descrever um tipo de domínio específico. Se um método muito poderoso (em termos conceituais) for escolhido, a especificação E_{Am} poderá tornar-se mais complexa do que o necessário devido à riqueza de instrumentos existentes ou à dificuldade de escolha entre as várias construções possíveis. Por outro lado, se um MESOO m muito pobre for utilizado, detalhes importantes de E_{Am} deixarão de ser descritos por não ter como representá-los no método de especificação. Como podemos ver na literatura [UML 99], a tendência é que haja sobreposição em termos de conceitos entre MESOOs. Com o surgimento de novos MESOOs e a atualização dos atuais, a tendência

é que a área de sobreposição aumente, como ilustra a Figura 4. Faz-se necessário, portanto, a sistematização de uma visão de equivalência entre os MESOOs. Esta poderá ser descrita com maior ou menor rigor formal. Neste último caso, uma possibilidade seria a construção de uma possível visão usando um esquema conceitual e sua população. Do ponto de vista de banco de dados, o esquema representaria o metamodelo e as especificações mapeadas para STL (descrições dos pacotes STL que populam o esquema) apresentariam um panorama de como os conceitos dos MESOOs são mapeados em termos de convergência de conceitos.

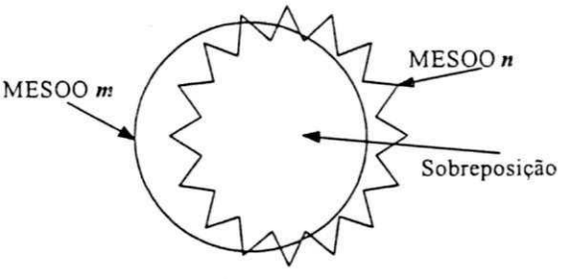

Figura 4: Sobreposição de Conceitos entre MESOOs

Uma proposta de construção de uma visão de banco de dados é apresentada em Medeiros *et al* [MEDE 96, 98]. Suponhamos que o processo de construir a especificação seja a elaboração de uma base de dados onde os conceitos STL são as entidades e as cláusulas STL correspondem aos relacionamentos ou atributos das entidades. A essência da comparação em STL depende da escolha dos seus conceitos básicos. Se um conjunto insuficiente de conceitos for usado para construir E_{mn}, a representação de MESOOs distintos pode ser reduzida à área de sobreposição como ilustra a Figura 4. No outro extremo, um conjunto excessivo de conceitos pode obscurecer importantes diferenças existentes entre os métodos. Neste contexto, a

análise e escolha do metamodelo STL é fundamental para representar os MESOOs de forma canônica [MEDE 00].

1.4 Objetivo da Tese

A indústria necessita empregar a reutilização de componentes mas isto não é possível sem o estabelecimento de padrões que garantam produtividade e qualidade [MEYE 2001]. A comunidade de desenvolvedores deve se preocupar mais com a maneira com que as empresas estão usando os MESOOs e o que constitui o ambiente "correto" (escolha do MESOO) [BASK 2001] que permita um melhor aproveitamento do legado. Estudar STL com estas preocupações em mente é o que motiva nossa pesquisa.

O centro de nosso trabalho está na proposição de um novo metamodelo para STL, que abstrai os principais conceitos dos diferentes MESOOs e, com isso, consegue uma representação canônica de pacotes STL. Conseqüentemente, as especificações desenvolvidas por métodos distintos poderão ser comparadas quanto a sua semelhança. Estudamos o Padrão IEEE 1175, no qual o metamodelo STL foi definido, para propor um novo conceito chamado de *Equivalência* que será utilizado para capturar diferenças e similaridades dos pacotes de informações que são transferidos entre ferramentas CASE baseadas em diferentes métodos.

Como podemos ver na literatura, existem vários Métodos de Especificação de Software Orientados a Objeto (MESOO) disponíveis no mercado. Cada MESOO pode ser subdividido em submodelos para oferecer ao desenvolvedor de software formas adequadas de apresentação do modelo da realidade do domínio do problema. Cada

MESOO pode produzir diagramas diversos para expressar determinadas características de uma visão ou perspectiva de interesse. A quantidade e tipo das perspectivas dependem de cada um dos MESOOs. Um exemplo encontra-se ilustrado na Figura 5. Note que, dependendo do diagrama do MESOO, a visão escolhida pode contemplar aspectos dinâmicos, através de tabelas de transição de estados, ou uma descrição funcional por meio de casos de uso. Os diagramas materializam os conceitos dos MESOOs por meio de símbolos gráficos ou descrição textual complementar mais apropriada. Apesar de existir a possibilidade de trabalhar com os MESOOs construindo especificações manualmente, a crise do software [SOMM 97] impõe que este processo seja automatizado.

Com a crescente demanda por software, surgem as ferramentas CASE que suportam os MESOOs e agilizam todo o processo de construção de programas. Há, como já discutimos, a necessidade de comunicação com transferência de semântica entre as ferramentas CASE de diferentes distribuidores, como ilustra a Figura 5.

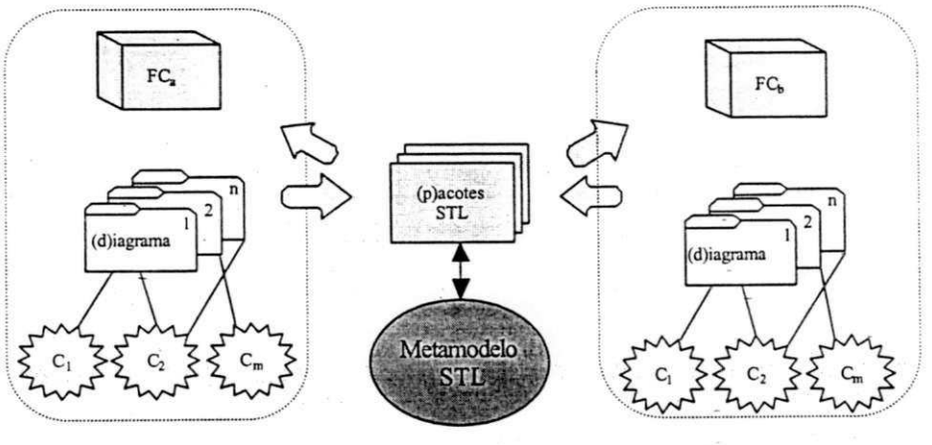

Figura 5: Comunicação entre Ferramentas CASE e um mesmo MESOO

STL original não se preocupa em registrar nenhuma informação sobre o mapeamento entre um MESOO m e um pacote STL p. Supondo Cm um conceito de m e $Cstl$ um conceito de STL, não há como obter informações sobre como Cm for mapeado para $Cstl$ ao longo da transmissão de vários pacotes. A granularidade da comunicação é sempre um pacote. Desta forma, esta tese contribui para a evolução do Padrão IEEE 1175 [IEEE 99] uma vez que propõe a criação de um metamodelo STL que permite a representação canônica dos MESOOs através de menos conceitos e maior capacidade de combinação entre eles. Por exemplo, sugerimos que os conceitos DataItem, DataKey, DataPart, DataRole, DataStore, DataType, e DataView sejam representados no nosso metamodelo proposto por um conceito base Entidade (Data) que possa ser combinado com outros conceitos derivados (relacionamentos).

STL está baseada nos conceitos apresentados na parte inferior da Figura 6 e o metamodelo composto de cinco conceitos principais e um conjunto de relacionamentos entre eles. Nossa proposta é alterar este metamodelo incluindo um *conceito de equivalência* para capturar as similaridades e diferenças entre os MESOOs durante o processo de transferência de pacotes STL $P = \{p_1, p_2, ..., p_n\}$ entre ferramentas CASE FC_{aA} e FC_{bA} $(a \neq b)$ da especificação de uma aplicação A.

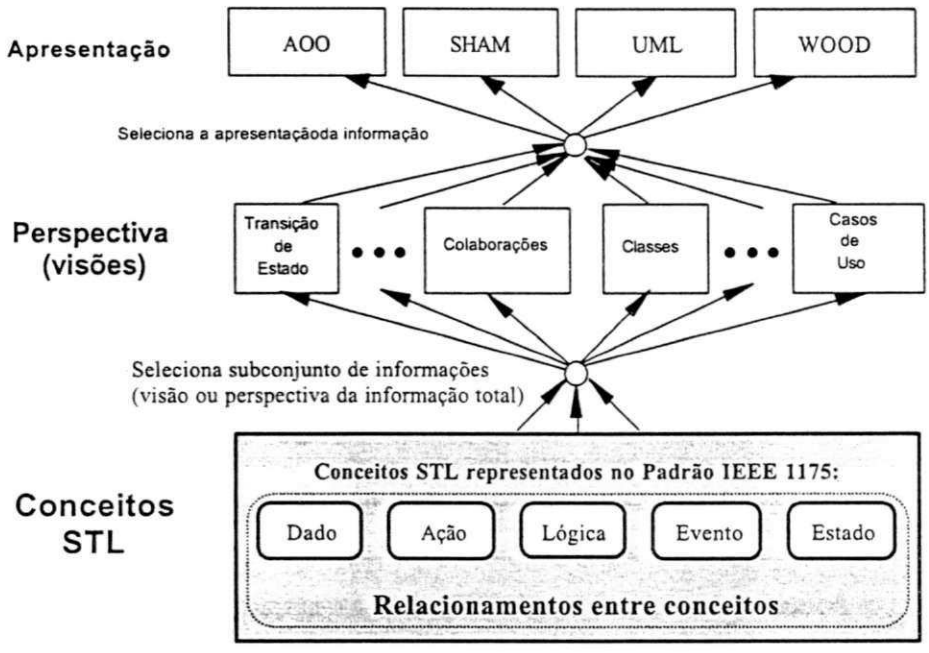

Figura 6: Visão Geral do Metamodelo de STL

Pretendemos utilizar a comunicação STL de pacotes P existentes entre as ferramentas CASE e propor uma alteração no conjunto de conceitos *base* e *derivados* (relacionamentos) STL, incluindo um novo conceito, chamado de *Equivalência,* para fazer o mapeamento entre conceitos C = $\{c_1, c_2, ..., c_n\}$ contidos em diagramas dos diferentes MESOOs m_1 e m_2 ^{o}M, $m_1 \neq m_2$, usados para representar a especificação de uma aplicação A e para fazer a transferência de semântica entre FC_{aA} e FC_{bA} de forma canônica, onde a e b representam dois fornecedores distintos, como ilustra a Figura 7. FC_{aA} e FC_{bA} podem suportar MESOOs distintos durante a comunicação.

O metamodelo de STL ocupa um papel importante na determinação do nível de detalhes a serem oferecidos para fazer o

mapeamento entre os conceitos e sua representação canônica em STL. Pretendemos ilustrar as atuais deficiências e apresentar um novo metamodelo para que STL represente os conceitos C e possamos ter uma visão de equivalência entre os MESOOs.

Pretendemos construir um interpretador de pacotes para interpretar pacotes eliminar as limitações ou deficiências do metamodelo atual de STL através do acréscimo de um novo conceito.

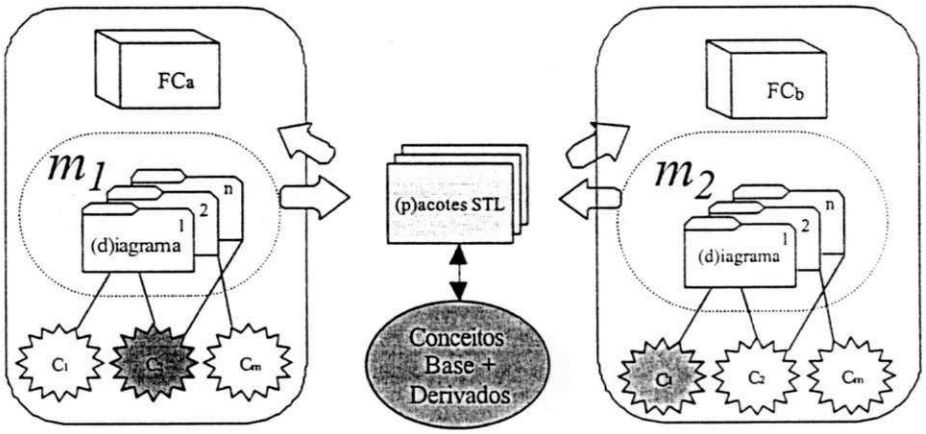

Figura 7: Visão de Equivalência entre MESOOs através de STL

1.5 Etapas da Tese

O desenvolvimento desta tese deu-se segundo as seguintes fases: especificação de um cenário que descreve situações do domínio de um problema a ser automatizado com solução informatizada a partir do uso de MESOOs. A identificação e descrição das funcionalidades é feita através de conceitos presentes nos MESOOs; aplicação dos métodos de especificação de software orientados a objetos ao cenário; identificação das dificuldades para transferir semântica entre ferramentas CASE (FC_{aA} e FC_{bA}, $a \neq b$) utilizando um

mesmo MESOO; apresentação de um novo "conceito" para oferecer mecanismos para comparação de MESOOs através de pacotes STL; acréscimo de um *conceito de equivalência* entre conceitos STL; Uso do novo metamodelo STL para transferir semântica entre Ferramentas CASE FC_{aA} e FC_{bA} produzidas por distribuidores independentes.

1.5.1 Especificação de um Cenário

Vamos utilizar a descrição de um cenário para ilustrar alguns conceitos e familiarizar o leitor com as abordagens empregadas na construção de uma especificação E_{Am} em cada um dos MESOOs $m \in M$. Estudaremos as principais notações dos MESOOs e apresentaremos alguns conceitos utilizados nos diversos diagramas na construção de especificações de software.

O cenário será usado para ilustrar as diferenças nas notações dos MESOOs e também as sobreposições em termos conceituais. Conceitos comuns como herança (generalização / especialização), agregação, associação e outros conceitos serão abordados para visualizar o mapeamento entre MESOOs e STL.

1.5.2 Aplicação dos MESOOs ao Cenário

Cada MESOO m é composto de uma notação com regras sintáticas e semânticas. A representação gráfica é distribuída entre diagramas $\{d\}$ que utilizam os conceitos fundamentais $\{c\}$ para construir uma especificação E_{Am} como ponto de partida para a informatização do sistema no cenário. Vamos construir E_{A1}, E_{A2}, E_{A3}, e E_{A4} utilizando os conjuntos de conceitos C_1, C_2, C_3, e C_4, respectivamente, para fazer o seu mapeamento para pacotes $\{P_1\}$, $\{P_2\}$, $\{P_3\}$ e $\{P_4\}$, em STL. Onde C_m, $m = 1, 2, 3$ e 4 são todos os

conceitos utilizados nos diagramas presentes na especificação da aplicação de interesse E_{Am}. P_m é o conjunto de pacotes utilizados para transferir E_{Am} entre ferramentas CASE que suportem m (FC_{aAm} e FC_{bAm}).

Como FC_{aA} e FC_{bA} são produzidas por diferentes fornecedores (a e b, sendo $a \neq b$) e, geralmente, apresentam formatos internos proprietários, mesmo utilizando um mesmo MESOO m, é necessário o uso de STL para fazer a comunicação. Isto será feito mediante o desenvolvimento de um interpretador de pacotes STL em função dos utilitários *lex* e *yacc*.

1.5.3 Identificação de Dificuldades para Transferir Semântica

Um determinado MESOO m, empregado no cenário descrito na seção anterior para criar especificações de uma aplicação E_{Am}, dependendo da notação definida em m, pode utilizar conceitos $C_m = \{c_1, c_2, c_3 ..., c_n\}$ de forma ambígua. Ou seja, para $i \neq j = 1,2, ..., n$, temos $\{c_i, c_j\}$ descrevendo uma mesma especificação E_{Am}. Em outras situações, podemos encontrar dificuldade de mapeamento envolvendo algum $c_i \circ C_m$ na construção de seus respectivos pacotes STL $p_i \circ P_m$.

Uma análise dos pacotes P_m será apresentada para ilustrar as limitações do metamodelo de STL e a necessidade de sua extensão ou modificação. Os conceitos base STL serão identificados e as restrições existentes entre os relacionamentos serão representadas pelos conceitos derivados.

Atualmente não há como verificar, em STL, se os pacotes p_m e p_n que refletem respectivamente, E_{Am} e E_{An}, construídas em função de MESOOs m e n, e utilizadas entre duas Ferramentas FC_{aA} e FC_{bA}

para descrever uma aplicação A, indiquem equivalência entre conceitos sendo transferidos. A automatização da BNF de STL, através dos comandos lex e yacc, pode ilustrar que pacotes são montados e desmontados isoladamente.

1.5.4 Proposta para Resolver Dificuldades/Problema

A solução do problema passa por uma reformulação do metamodelo STL e a inclusão de um novo conceito que permita manter memória do mapeamento entre os conceitos dos MESOOs e STL. O novo metamodelo deverá permitir que, a um determinado nível de detalhes, as especificações $E_{A,m}$ sejam descritas através de pacotes P_m de forma canônica. Estamos apresentando proposta de inclusão do conceito de equivalência no metamodelo STL para representar os conceitos C_m dos MESOOs selecionados para esta tese[COAD 90, 91, SHLA 88, 91, UML 96, 99, WIRF 90], onde $M = \{"AOO", "SHAM", "UML", "WOOD"\}$.

As ferramentas CASE possuem um formato proprietário de representação das especificações das aplicações $E_{Am,}$. Cada ferramenta pode suportar um ou mais MESOOs. Ferramentas CASE que utilizem os pacotes STL P_m poderão explorar melhor as características individuais de métodos distintos m e n na construção de uma especificação E_{Ax} que utiliza características específicas de dois MESOOs m e n.

Esta possibilidade de uso de múltiplas linguagens para construir uma única especificação, usando diferentes MESOOs existentes, pode ser usada para resolver problemas de evolução das MESOOs nas empresas, onde novos métodos de especificação precisam conviver com outros durante uma fase de transição.

Estamos propondo um metamodelo em termos de conceitos STL básicos e derivados e a inclusão do conceito de equivalência entre conceitos dos MESOOs.

1.5.5 Acrescentando Conceitos à STL

A partir da identificação em 1.5.4, propomos uma revisão dos conceitos base e derivados para contemplar a visão de equivalência entre as representações em STL *eq* dos conceitos C_m e C_n dos MESOOs $m, n \in M$ e a criação de uma biblioteca de conceitos canônicos que permita identificar equivalência ou sobreposição durante o mapeamento entre conceitos dos MESOOs.

1.5.6 Uso de STL para Transferir Semântica

Seja E_{Am} uma especificação de uma aplicação A escrita em um MESOO $m \in M$ e Fc_a e Fc_b duas ferramentas CASE distintas a e b. Desenvolveremos um interpretador de pacotes I_{stl} que operacionalize a transferência de E_{Am} entre Fc_a e Fc_b.

Seja I_{stl} o interpretador de pacotes STL construído em função do metamodelo de STL e usado como mecanismo de transferência. Assim, por definição de STL teremos que

$$I_{stl}(Fc_a(E_{Am})) = I_{stl}(Fc_b(E_{Am})).$$

Ou seja, uma mesma Especificação E_{Am} é vista da mesma forma, mesmo que usando ferramentas CASE distintas, através do metamodelo STL.

Podemos usar ainda a visão de equivalência envolvendo o mapeamento dos conceitos C_m, C_n, de métodos distintos e I_{stl}, I_{stl} é definido em funções dos conceitos STL no metamodelo proposto.

Uma análise destes últimos conceitos vai possibilitar a visão de equivalência em termos de sobreposição de conceitos entre MESOOs *m* e *n* em termos de

$$I_{stl}(Fc_a(E_{Am})) \approx I_{stl}(Fc_b(E_{An})),\ m \neq n.$$

As ferramentas Fc_a e Fc_b se comunicarão através de um interpretador de pacotes I_{stl} posibilitando a visualização de conceitos equivalentes em MESOOs distintos.

1.6 Resumo das Contribuições

Existe um consenso tanto na indústria como na academia de que a fase de especificações é uma das mais importantes no processo de desenvolvimento de software. Erros encontrados nesta fase podem ser corrigidos com custo muito baixo se comparados com as fases finais de codificação e testes [SOMM 97].

O crescimento do número de MESOOs e a necessidade de se trabalhar com mais de um método de especificação de software que mais se adapte às condições ambientais exigidas levam à necessidade de escolha constante de que classe de ferramentas (conceitual e operacional) usar frente aos diversos problemas da vida real.

O metamodelo de STL preocupa-se, basicamente, em transmitir informações através de pacotes. A construção de uma biblioteca de funções que permita o uso de STL, através da *linkedição* por parte dos desenvolvedores de ferramentas CASE de diferentes distribuidores — eles terão que incorporar a funcionalidade de empacotamento de STL de forma explícita—, e a proposta de um novo metamodelo para a Linguagem STL, que inclui o conceito de *Equivalência*, são as principais

contribuições de nosso trabalho. Propomos usar STL como base para avaliar como as especificações dos MESOOs são mapeadas. Através dos resultados desta operação oferecemos subsídios para comparação entre MESOOs.

Nossa contribuição para o Padrão IEEE 1175 [IEEE 92-99] tem sido no sentido de advogar a criação de um metamodelo STL que permita a representação canônica dos MESOOs através de menos conceitos e maior capacidade de combinação entre eles [MEDE 00]. Por exemplo, sugerimos que os conceitos *DataItem*, *DataKey*, *DataPart*, *DataRole*, *DataStore*, *DataType*, e *DataView* sejam representados no nosso metamodelo proposto por um conceito base Entidade (Data) que possa ser combinado com outros conceitos derivados (relacionamentos). Propomos a inclusão de um conceito de *Equivalência* em STL que permita identificar os conceitos equivalentes entre os MESOOs.

Adicionalmente, esperamos contribuir com a discussão que leve à construção de formas mais objetivas de treinar pessoas em mais de um MESOO. Na abordagem convencional, cada MESOO *m* é estudada separadamente. Já, utilizando um metamodelo como o proposto nesta tese, os MESOOs são estudados com o objetivo de verificar se os conceitos base e os derivados, mapeados nos seus respectivos modelos, são suficientes para representar a notação e a semântica de cada *m*. Apesar de não ser o foco central de nosso trabalho, a identificação e classificação dos conceitos fundamentais dos MESOOs geram uma perspectiva crítica estabelecendo parâmetros que favorecem o aprendizado ou a análise de novos MESOOs.

Esperamos que nossa tese motive desenvolvedores de ferramentas CASE a se preocuparem mais com os aspectos de padronização e comunicação entre módulos autônomos em um ambiente cooperativo. Visualizamos, no futuro, uma possibilidade de reaproveitamento de artefatos de software através de uma abordagem que utilize metamodelos que permita a representação e o mapeamento entre conceitos. Neste sentido, a PRG (Proposta de Representação Gráfica) é uma contribuição deste trabalho para ilustrar mapeamentos de conceitos para uma representação universal e possível reaproveitamento de especificações legadas.

A necessidade de formalização do conceito de equivalência fez com que construíssemos uma proposta de Proposta de Representação Gráfica (PRG) que não deixa de ser uma outra notação. Entretanto, como pudemos observar ao ministrar cursos de análise e projetos de sistemas, onde vários MESOOs são discutidos, uma abordagem de ensino baseada na identificação dos principais conceitos das metodologias e o seu respectivo mapeamento para a PRG mostrou-se mais eficiente para o aprendizado de novos métodos.

1.7 Organização da Tese

Além desta introdução, este trabalho é composto de outros cinco capítulos. O foco central de nossa pesquisa encontra-se no quarto e quinto Capítulos. Descrevemos a seguir como esta tese está organizada.

No Capítulo 2, apresentaremos um cenário onde aparecem diagramas representando conceitos presentes em todos os MESOOs, como herança, associação e agregação. Discutiremos os principais aspectos dos MESOOs mostrando sua sintaxe e semântica. Este capítulo é para familiarizar o leitor com as notações utilizadas pelos métodos de especificação de software orientado a objetos.

No Capítulo 3, apresentaremos uma visão geral de STL, seu metamodelo e a semântica para transferência de pacotes e discutimos suas limitações com relação à equivalência de conceitos que estão sendo transferidos entre ferramentas CASE.

No Capítulo 4, apresentamos uma proposta de notação gráfica para STL. Nossa intenção é visualizar graficamente os conceitos dos MESOOs em uma forma canônica em STL.

No Capítulo 5, vamos apresentar o metamodelo proposto, incluindo o novo conceito de *Equivalência*, para representar os conceitos dos MESOOs e analisar quais os benefícios de mapeá-los de forma canônica.

Os resultados de nossa pesquisa são apresentados no último Capítulo "Conclusão e Sugestões para Trabalhos Futuros". Finalmente, os Anexos A, B e C são reservados para apresentar, respectivamente, detalhes da implementação do interpretador de pacotes STL, apresentar uma visão geral de como uma comparação dos MESOOs pode ser feita em função das cláusulas STL; ilustrar o código fonte das Ferramentas A4O.

Apresentamos a seguir uma visão geral dos principais conceitos dos MESOOs de interesse desta tese.

Capítulo

2

Visão Geral dos MESOOs e o Cenário de Interesse

Neste capítulo, descrevemos uma situação ou "cenário" do mundo real onde um problema precisa ser modelado para a construção de um software que automatize a sua solução. O cenário não será exaustivamente explorado nos vários Métodos de Especificação de Software Orientados a Objetos (MESOOs [COAD 90, 91], [SHLA 88, 91], [WIRF 90] e [UML 96-2000]). Nossa intenção é familiarizar o leitor com as diferentes notações usadas para representar parcialmente o domínio do problema. Vamos identificar os principais conceitos dos MESOOs e propor um cenário que ilustre o uso de pacotes STL como mecanismo para identificar sobreposições e diferenças conceituais existentes entre os MESOOs.

2.1 Descrição do Problema

Suponha que existe uma transportadora que deseja automatizar o recebimento e a entrega de encomendas. Existem

caminhões, carros de passeios e motos. As encomendas devem ser recebidas e classificadas de acordo com suas dimensões e peso, levando-se em consideração os meios de transporte disponíveis e sua capacidade de lotação. A seleção deve levar em conta o tempo máximo de espera associado a cada tipo de transporte e priorizar a população dos veículos cujo *timeout* está para expirar. Um roteiro das ruas a serem percorridas deve ser emitido para o respectivo motorista que se encontra no depósito. Uma ordem deve ser emitida para o motorista iniciar a entrega quando encher um transporte ou decorrer um tempo previamente estabelecido para cada um dos transportes.

Uma área comum é utilizada pelos veículos e deve ser controlada por um portão que é aberto apenas para o transporte liberado para iniciar a entrega. É preciso controlar o recebimento e o envio de encomendas envolvendo caminhões, carros de passeio e motocicletas. Os motoristas podem trafegar por várias vias mas devem prestar atenção à sinalização que controla a abertura e fechamento dos portões. As encomendas devem ser selecionadas à medida que forem chegando. O armazenamento deve levar em consideração os transportes que estão disponíveis e estão aguardando completar uma carga para iniciar a entrega.

Conforme já falamos anteriormente, estamos preocupados com os Métodos de Especificação de Software Orientados a Objetos. Assim, usaremos fragmentos do cenário acima para ilustrar cada uma das metodologias que explicitam o processo de uso dos MESOOs.

Apresentaremos, a seguir, uma visão geral de cada MESOO a título de ilustração das principais atividades descritas nas metodologias

que explicitam os seus processos de utilização. O objetivo é situar o leitor para identificar conceitos relevantes a serem mapeados para STL.

2.2 Visão Geral do Método de Coad e Yourdon

O método de Análise Orientada a Objetos (AOO) de Coad e Yourdon aplica-se tanto à análise quanto ao projeto de sistemas. Foram publicados dois livros [COAD 90, 91] nos quais maiores informações podem ser encontradas.

O método herda a ferramenta de modelagem conceitual Entidade-Relacionamento do mundo dos bancos de dados. A descrição de um modelo que represente características estáticas e dinâmicas do domínio do problema é proposta. O processo é descrito em termos das seguintes atividades:

- Identificar classes e objetos;
- Identificar estruturas;
- Identificar assuntos;
- Definir atributos;
- Definir serviços;
- Projetar os componentes do sistema.

2.2.1 Identificação de Classes e Objetos

A identificação das classes e objetos é feita através da análise do domínio do problema por meio de entrevistas, observações e

outros documentos ou formulários disponíveis. Como podemos observar (vide Figura 8), existe uma representação para classes e objetos e outra representação para classes, quando se tratar de classes abstratas.

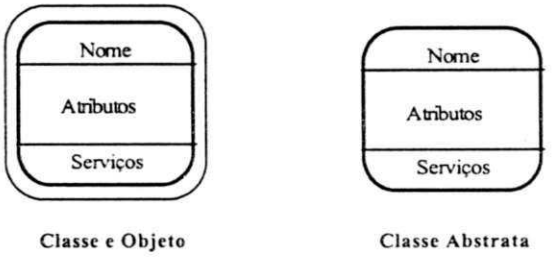

Figura 8: Classes e Objetos e Classes Abstratas

O resultado da atividade de identificação de classes e objetos demanda a produção de uma descrição do problema. Como nosso objetivo aqui é mostrar uma visão geral dos elementos mais importantes de cada método, vamos supor o seguinte problema para ilustrar os exemplos sobre os conceitos dos métodos de especificação de software orientado a objetos. Vamos supor que seja necessário "controlar o recebimento e envio de encomendas envolvendo caminhões, carros de passeio e motocicletas. Os motoristas devem trafegar por várias vias e prestar atenção à sinalização que controla a abertura e fechamento dos seus respectivos portões. As encomendas são selecionadas à medida que vão chegando. Deve-se levar em consideração os veículos disponíveis para completar uma carga e iniciar a entrega."

Dado o problema, os autores do método sugerem encontrar substantivos para a identificação de objetos. Vamos usar transporte,

na Figura 9, como uma generalização para carro, motocicleta e caminhão, ilustrando a representação de classes abstratas.

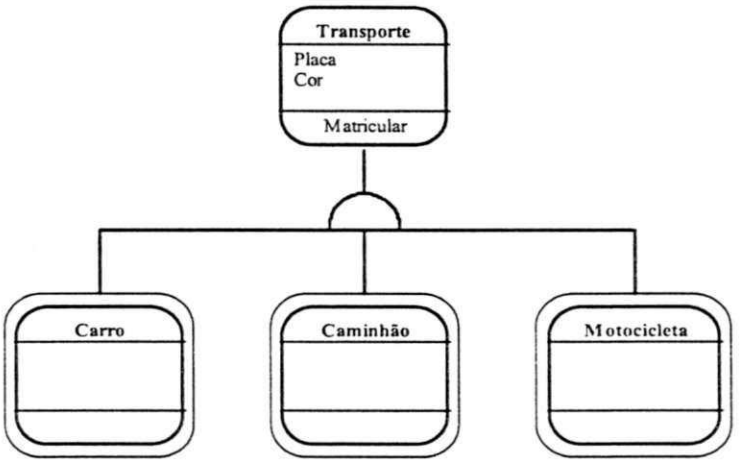

Figura 9: Classes Concretas e Abstratas

2.2.2 Identificação de Estruturas

A identificação de estruturas é uma forma de encontrar os agrupamentos de como as classes se relacionam, através de agregação e generalização. Os objetos da Figura 9 podem ser combinados através de generalização e especialização. A classe transporte generaliza informações das motocicletas, caminhões e carros. Podemos ainda usar objetos para descrever relacionamentos onde o todo é formado pelas partes. A Figura 10 mostra um exemplo em que os meios de transporte podem ser uma motocicleta, um carro ou um caminhão, onde estes herdam características de transporte (possuir placa, cor, ter que registrar o veículo no Departamento de Trânsito, etc). Por sua vez, transporte é composto das partes motor e carroceria (agregação).

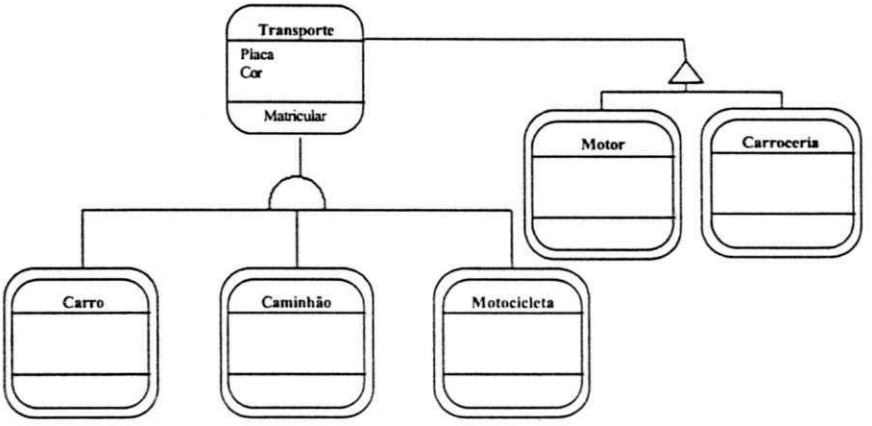

Figura 10: Herança e Agregação em Coad/Yourdon

2.2.3 Identificação de Assuntos

Identificação de assuntos é uma forma de agrupamento das classes e objetos em blocos. É uma forma de modularizar o modelo de objetos. Para exemplificar, podemos seguir uma regra dada pelo autor [COAD 91]— usar o nome de uma classe que é a base de uma generalização ou agregação como identificador do assunto—, e batizar o assunto "transporte" fazendo referência à Figura 10. Outro exemplo poderia ser o assunto "estacionar" envolvendo as classes caminhão e portão, assumindo que a primeira deva solicitar que um determinado portão de uma garagem seja aberto, como está ilustrado na Figura 11.

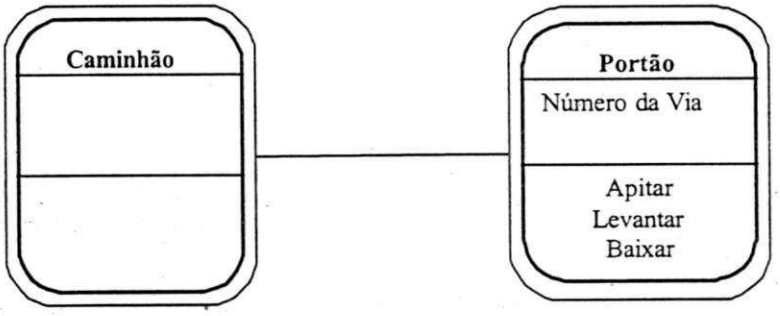

Figura 11: Assunto Estacionar Caminhão

2.2.4 Definição de Atributos

A definição de atributos explicita que características das classes de objetos estão sendo consideradas. O autor recomenda que atributos compostos sejam quebrados em elementos atômicos. Na classe Portão, por exemplo, o número da via é um atributo.

2.2.5 Definição de Serviços

Os serviços representam o comportamento dos objetos. Ações como "levantar", "baixar" e "apitar" são exemplos de serviços do objeto portão na Figura 11. A indicação de como os serviços são solicitados está ilustrada na Figura 12. Uma seta indica o sentido da mensagem que um objeto envia para o outro.

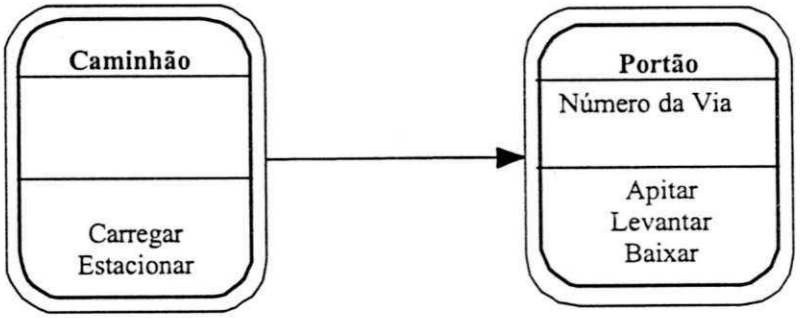

Figura 12: Comunicação entre Objetos

2.3 Visão Geral do Método de Shlaer/Mellor

O Método de Shlaer/Mellor [SHLA 88] baseou-se, primeiramente, na modelagem da informação como recurso principal para identificar objetos e classes no domínio do problema. Depois, em Shlaer-Mellor [SHLA 91], a abordagem adotada foi mais voltada para a modelagem do comportamento dos estados dos objetos e o controle do processo em que estas ações eram executadas.

Podemos aplicar o método seguindo as seguintes atividades:

- Identificar Objetos;
- Identificar Atributos;
- Identificar Associações;
- Identificar Comportamento.

2.3.1 Identificando Objetos

Na realidade, nesta fase Shlaer-Mellor [SHLA 88, 91] vai identificar as classes de objetos que são importantes para o domínio do problema. Como no método anterior, este processo é feito através da análise do domínio do problema por meio de entrevistas, observações, e outros documentos ou formulários disponíveis.

O método estabelece que sejam procuradas coisas tangíveis do mundo real, uma função ou um papel, uma ocorrência de um evento ou uma interação. Por exemplo, as classes de objetos caminhão e portão pertencem ao mundo real e são tangíveis; já a pessoa que dirige o caminhão exerce esta função ou papel; a chegada

de uma solicitação de carga é um evento; e finalmente uma interação normalmente vai envolver dois ou mais objetos do modelo como a solicitação do objeto transporte para baixar o portão.

Supondo que o problema é "controlar o recebimento e envio de encomendas envolvendo caminhões, carros de passeio e motocicletas, os motoristas devem trafegar por várias vias e prestar atenção à sinalização que controla a abertura e fechamento dos seus respectivos portões. As encomendas são selecionadas, à medida que vão chegando. Deve-se levar em consideração os transportes disponíveis para completar uma carga e iniciar a entrega", podemos identificar alguns objetos candidatos:

> Coisas tangíveis: encomenda, carro, caminhão, motocicleta, portão, sinal;
>
> Papel: motorista, recepcionista;
>
> Eventos: carga pronta, início da entrega, portão abriu, portão fechou;
>
> Interação: solicitar fechamento, trafegar, carregar.

Após a identificação dos objetos candidatos, temos que verificar a uniformidade, ou as suas características comuns, ou se o objeto não se resume a apenas um nome e deve ser eleito como uma característica ou comportamento de outro objeto. O autor sugere que seja criada uma relação contendo a descrição dos objetos selecionados, semelhantes a um dicionário de dados na abordagem estruturada. A representação gráfica dos objetos está descrita na Figura 13. Observe

que há uma numeração e a sugestão para se colocar a inicial do nome da classe entre parênteses.

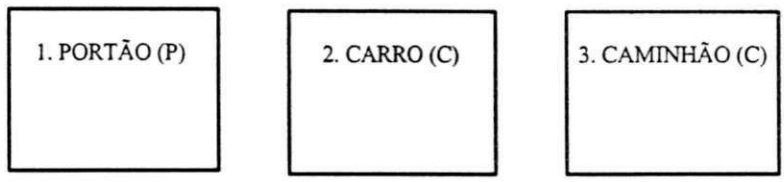

Figura 13: Representação de Objetos

2.3.2 Identificando Atributos

Os *atributos* são usados para rotular ou nomear instâncias. Os atributos descritivos são empregados e descrevem fatos que são inerentes a todas as instâncias de um objeto. O atributo referencial é utilizado para descrever o relacionamento das instâncias dos objetos.

Podemos observar na Figura 14 alguns objetos do domínio do problema e seus respectivos atributos. Estes vêm sempre abaixo do nome do objeto. Note que este método utiliza conceitos da teoria de banco de dados relacionais quando sugere que os atributos sejam normalizados e que uma indicação "*" deve ser colocada junto aos atributos "chave". Os atributos também devem ser descritos no "dicionário" de dados, da mesma forma que foram os objetos.

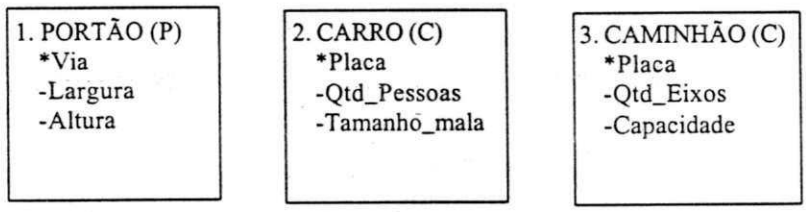

Figura 14: Objetos com Atributos

2.3.3 Identificando Associações

Esta atividade está ligada ao processo de identificação e definição dos relacionamentos ou do conjunto de associações que existem nos objetos do domínio do problema. Um exemplo pode ser visto na Figura 15. Cada associação deve ser avaliada em termos de sua cardinalidade "1:1" (↔), "1:m" (↔>), e "m:n" (<↔>). Observe que os atributos referenciais formalizam o relacionamento e podem ser melhor detalhados e referenciados pelos códigos *R1*, *R2* e *R3*. Podemos ter objetos descrevendo associações m:n quando for necessário introduzir alguma característica ou comportamento ao relacionamento. Observe que o relacionamento do tipo "é parte de" descreve uma agregação.

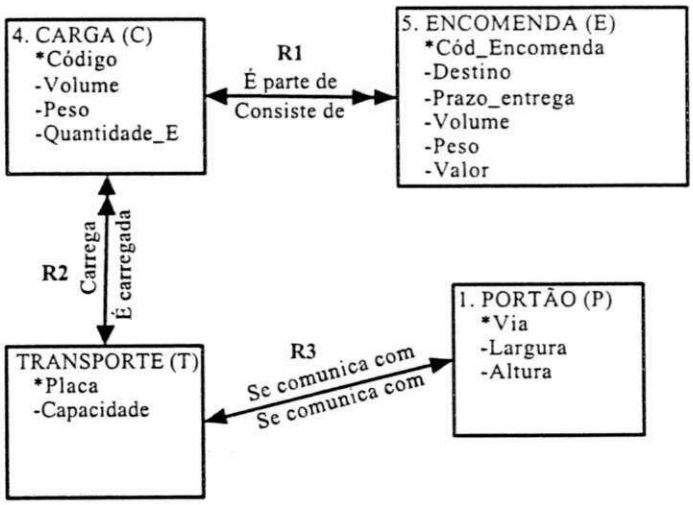

Figura 15: Diagrama da Estrutura da Informação

2.3.4 Identificando Comportamento

O comportamento dos objetos é feito através de máquinas de estado. Cada estado é numerado e rotulado, como podemos ver no

exemplo da Figura 16. As ações que ocorrem em um estado são mostradas abaixo do estado (em pseudo-código) e as transições são rotuladas com seus respectivos eventos. Os eventos (E1, E2, ...) afetam as instâncias dos objetos que os recebem. Podemos observar o

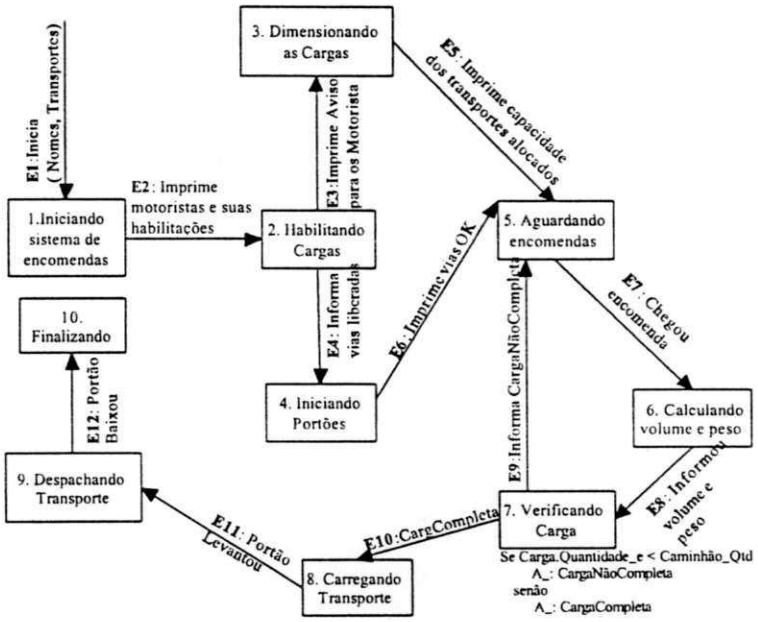

Figura 16: Máquina de Estado de Shlaer/Mellor

ciclo de vida para a montagem e despacho de uma carga. Inicia no estado 1 com o fornecimento da informação de quais motoristas e transportes encontram-se disponíveis e termina no evento 10 com o envio da carga para seu destino. A seqüência de eventos é auto-explicativa e a máquina vai mudando de estado à medida que os eventos vão ocorrendo. No estado 7 (Verificando Carga) pode-se ver o pseudocódigo executado para decidir quais dos eventos seguintes ocorrerão.

Há uma recomendação dos autores do método para que as considerações dos caminhos que levem ao tratamento de erros ou

falhas sejam postergadas para depois de serem definidos os percursos normais. Há ainda a sugestão para que relacionamentos que envolvam competição sejam modelados com uma máquina de estado à parte.

2.3.4.1 Tabela de Transição de Estados

Outra forma de descrever o comportamento é através de uma *Tabela de Transição de Estados* As linhas da tabela representam os vários estados que um objeto pode assumir e as colunas representam os eventos que um objeto pode receber. Quando um objeto recebe um evento em um determinado estado pode ocorrer uma transição ou o próprio objeto pode ignorar o evento.

2.3.4.2 Lista de Eventos

A *Lista de Eventos* é outro mecanismo disponível para descrever as fontes e destinos de todos os eventos juntamente com os dados transmitidos. A Tabela 1 mostra uma visão parcial para o nosso problema da transportadora. Podemos explicitar as entidades externas como o operador e os clientes que vão usar o sistema, para justificar a fonte e destino de alguns eventos.

Tabela 1: Lista de Eventos Shlaer-Mellor

Evento	Significado	Dado do Evento	Fonte	Destino
E1	Inicia Sistema	Relação motorista, transportes	Operador	Sistema
E2	Imprime habilitação do motorista	Classe do motorista	Motorista	Sistema
...	...	...	...	...

2.3.4.3 Modelo de Comunicação dos Objetos

Outro recurso existente para modelar o comportamento e ajudar a dar uma visão geral do sistema é o *Modelo de Comunicação de*

Objetos. Os eventos entre os objetos são mostrados identificando as camadas ou principais blocos e os eventos que podem ocorrer entre eles. Na Figura 17, por questão de limitação de espaço, resolvemos ilustrar apenas os eventos de interesse. Explicitamos o cliente como parte do modelo e sua interação com o módulo de recepção.

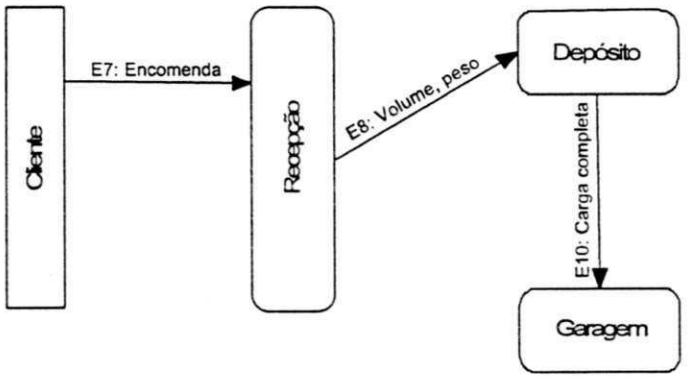

Figura 17 : Diagrama de Comunicação de Objetos Shlaer/Mellor

2.3.4.4 Diagrama de Controle de Execução

Ações e eventos encontram outra forma de representação no *Diagrama de Controle de Execução.* Cada ação tem dois temporizadores associados (o tempo da ação e um tempo de espera para completar a transição para o próximo estado). O tempo da ação é colocado internamente ao objeto, como podemos ver na Figura 18. O tempo de espera pode ser omitido quando não for crítico para a realização da operação.

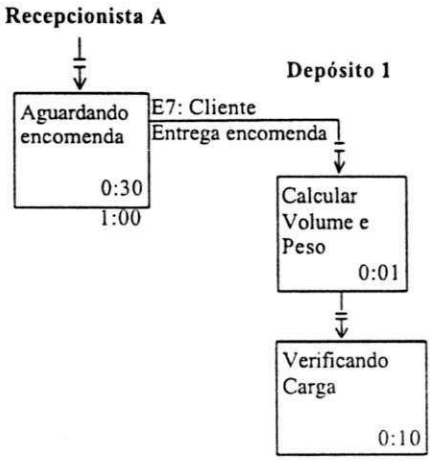

Figura 18: Diagrama de Controle de Execução de Shlaer/Mellor

2.3.4.5 Diagrama de Fluxo de Dados das Ações

Este conceito data da década de 70 e foi introduzido por Coad e Yourdon. A modificação introduzida foi para a representação do controle que pode ser feita de duas formas: explicitamente através de fluxo de controle ou de forma implícita por meio de certos fluxos de dados. Como pode-se observar na Figura 19, a seta pontilhada indica um controle para saber quando a carga atual for completada.

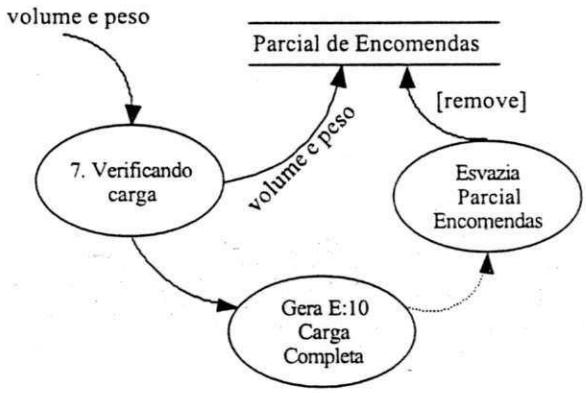

Figura 19: Diagrama de Fluxo de Dados das Ações Shlaer/Mellor

2.4 Visão Geral do Método de Wirfs-Brock

O Método de especificação de software orientado a objetos desenvolvido por Rebeca Wirfs-Brock, Brian Wilkerson, e Lauren Wiener [WIRF 90] é baseado na identificação de responsabilidades e cooperações entre objetos, com ênfase nos aspectos dinâmicos ou comportamento dos objetos.

As seguintes atividades são descritas para a análise de um sistema:

1. Identificação de classes e sua hierarquia;
2. Identificação de responsabilidades;
3. Identificação de colaborações;
4. Definição de contratos;
5. Especificação de protocolos.

2.4.1 Identificação de Classes e Hierarquia

As classes são determinadas a partir do domínio do problema. Como nos métodos anteriores, este processo pode se basear em observações *in loco*, entrevistas e documentos disponíveis para determinação das necessidades do sistema. Um documento ou descrição do problema deve ser construído.

O conjunto inicial de classes é selecionado do domínio com base nos nomes que aparecem na descrição do problema. Tomando como base a descrição usada nos métodos das seções anteriores, os nomes grifados candidatos a classes seriam: "controlar o recebimento e envio de encomendas envolvendo caminhões, carros de passeio e motocicletas. Os motoristas devem trafegar por várias vias e prestar

atenção à sinalização que controla a abertura e fechamento dos seus respectivos portões. As encomendas são selecionadas, à medida que vão chegando. Deve-se levar em consideração os veículos disponíveis para completar uma carga e iniciar a entrega."

O autor sugere as seguintes perguntas para determinar se os nomes serão classes do domínio: a) o nome é um objeto físico do escopo do problema? Em caso afirmativo, é um forte candidato a classe; b) é uma entidade conceitual do domínio do problema? Neste caso, é também uma informação importante e o nome pode ser eleito para ser uma classe; c) o nome é uma interface com o ambiente externo? Sim, se ele tem comunicação com o mundo; d) o nome é um atributo de um objeto já existente?

Os autores sugerem que os objetos com características comuns sejam classificados e as classes anotadas individualmente em fichas como mostra a Figura 20. A indicação da classe carro é seguida pela indicação de sua superclasse transporte. É uma forma de registrar os relacionamentos entre classes, subclasses e superclasse.

Classe: Carro	
Transporte	

Figura 20: Fichas para Anotação de Classes Individuais de Wirfs-Brock

A Figura 21 descreve, graficamente, a forma de representar herança entre as classes transporte, carro, caminhão e motocicleta. Há

uma indicação na classe transporte de que ela é abstrata. Ou seja, ela não pode ser instanciada.

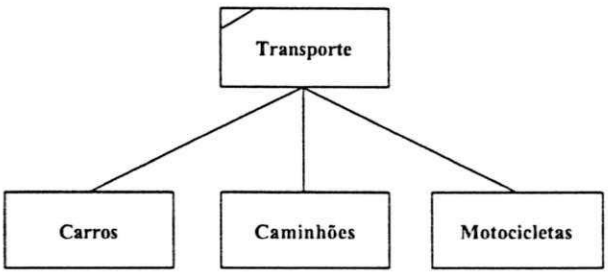

Figura 21: Representação de Herança em Wirfs-Brock

Alternativamente, outra forma de representação gráfica da hierarquia é através do Diagrama de Venn, como pode ser visto na Figura 22.

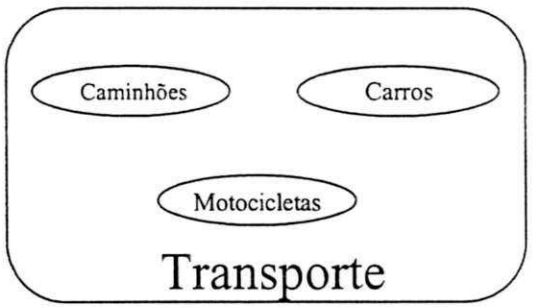

Figura 22: Hierarquia através do Diagrama de Venn

2.4.2 Identificação de Responsabilidades

As responsabilidades indicam as ações que um objeto pode tomar, seu comportamento e o conhecimento que ele deve manter. As fichas de anotações das classes, como podemos ver na Figura 23, devem conter uma descrição resumida das responsabilidades. Estas poderão ser detalhadas de forma textual ou por intermédio de um dicionário de dados implementados em um repositório central através de uma ferramenta CASE.

Classe: Portão	
Sabe quais vias estão funcionando	

Figura 23: Classe com Responsabilidade

2.4.3 Identificação de Colaborações

As *colaborações* mostram como os objetos das classes interagem. É um mecanismo onde existe um cliente e um servidor associado às responsabilidades dos objetos. Pode-se usar as fichas, conforme a Figura 24, para registrar quais serviços serão providos para as classes clientes. Aqui a classe portão oferecerá o serviço "levantar". Outra forma de visualização é feita por meio da Representação de Colaboração usando Generalização e Especialização para ilustrar herança entre as classes, como podemos ver na Figura 25. Observe que a classe transporte é uma generalização para carros, caminhões e motocicletas. A representação da colaboração "levantar" está ilustrada na Figura 26. A classe *carro* solicita um serviço da classe *portão*. Este serviço estará detalhado através de um contrato, descrito na seção a seguir.

Classe: Portão	
Sabe quais vias estão funcionando	
Mantém ordem de prioridade nos pedidos	Levantar

Figura 24: Ficha de Classe com Colaboração

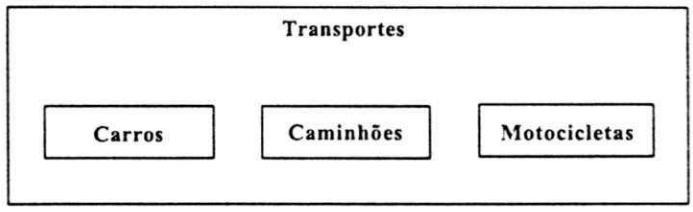

Figura 25: Representação de Generalização / Especialização

2.4.4 Definição de Contratos

A identificação de *contratos* serve para agrupar responsabilidades para melhorar a compreensão do projeto. Um contrato define um conjunto de requisições que uma classe cliente pode fazer a uma classe servidora. Os contratos são representados por semicírculos e numerados para poder serem referenciados e detalhados em outras partes do método. A seta indica a classe que solicita o serviço. No caso da Figura 26, *carro* solicita que o portão seja levantado.

Figura 26: Colaboração através de Contrato

2.4.4.1 Subsistema

Subsistema é uma forma de esconder ou empacotar informações. A representação de um subsistema pode ser vista na Figura 27. Os subsistemas são representados por retângulos com os cantos arredondados e os números dos contratos servem também de interface para os subsistemas.

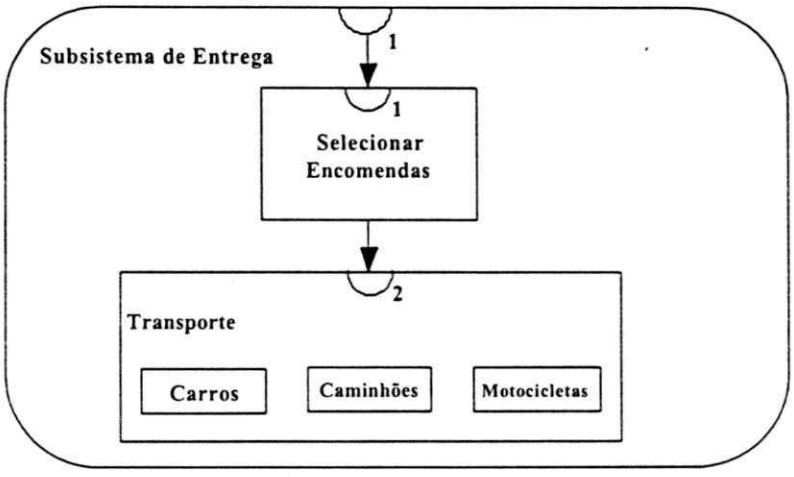

Figura 27: Colaboração Através de Contratos

Aqui podemos ver parte de um subsistema responsável pela seleção de encomendas e escolha do tipo de transporte a usar.

2.4.5 Especificação de Protocolos

A *especificação de protocolos* é um processo definido para refinar as responsabilidades distribuídas às classes. Os autores definem protocolos como um conjunto de assinaturas que as classes vão responder. Na realidade, os protocolos são uma implementação dos contratos. Os protocolos vão indicar como as classes devem se comportar baseadas na formalização de contratos através da definição dos métodos e suas assinaturas (o nome do método, o tipo dos seus parâmetros, e o seu tipo de retorno).

2.5 Visão Geral do Método UML

Apresentaremos nesta seção uma visão geral dos principais conceitos e ferramentas de modelagem conceitual de UML ("*Unified Modeling Language*").

UML é um método[11] de especificação de software orientado a objetos que engloba vários conceitos dos MESOOs já existentes no mercado. Ele herda características de outros MESOOs como OMT [RUMB 91], Booch [BOOC 91] e OOSE [JACO 92].

A criação de um método que fundisse ou suportasse vários conceitos já consagrados, já havia sido tentada anteriormente pelo Método *Fusion* [COLE 94] sem o sucesso apresentado por UML [UML 99]. O fato de a proposta de criação de UML ter partido de Grady Booch, James Rumbaugh, e Ivar Jacobson, autores de métodos consagrados [BOOC 91, RAMB 91, JACO 92] e aliados a uma estratégia aberta de evolução do modelo, pode ser a razão da atual aceitação de UML.

Houve uma preocupação para não se definir o processo ou as atividades de aplicação do método de especificação de software orientado a objetos em UML. Ou seja, escreveu-se uma linguagem ou notação contendo apenas os principais conceitos dos MESOOs previamente existentes. A metodologia não foi definida e só depois surgiu o processo unificado de desenvolvimento de software[JACO 99, 99a]. Esta estratégia teve como objetivo juntar um maior número

[11] Observe que UML não define o processo, a metodologia de uso de sua notação. Há uma preocupação em definir apenas a notação gráfica com sua respectiva semântica.

de voluntários que contribuíssem com a evolução da linguagem e conseguir focar apenas a notação e não o processo.

Em UML os conceitos centrais podem ser divididos em visões da realidade, cada uma representada por um conjunto de diagramas. Este artifício ajuda a lembrar que diagramas usar dependendo do foco ou interesse em modelar características específicas de uma aplicação.

2.5.1 Visões e Diagramas

As *visões* servem para mostrar os diferentes aspectos do sistema. Cada visão pode ser composta por um ou mais diagramas. As visões são as seguintes:

- Visão de Casos de Uso;
- Visão Lógica;
- Visão Dinâmica ou Concorrente;
- Visão de Componentes.

2.5.2 Visão de Casos de Uso

A *visão de caso de uso* define dois conceitos básicos: o de *ator* e o de *caso de uso*. O primeiro identifica as *entidades* importantes do domínio do problema. Os *caso de uso* ilustram, como se fossem uma foto, os contextos ou relacionamentos onde os atores se relacionam. São os exemplos de uso envolvendo os atores de uma parte do domínio do sistema que se deseja modelar.

O poder dessa ferramenta de modelagem conceitual reside na sua simplicidade. É através dos conceitos presentes no Diagrama de

Casos de Uso que as funcionalidades, que o sistema deve implementar, são descritas do ponto de vista do cliente ou do usuário externo. Os *casos de uso* servem ainda para direcionar o desenvolvimento de outras visões, e são úteis também para validar o sistema, à medida que usuários e a equipe técnica de desenvolvimento utilizem uma forma simples e eficaz de comunicação.

Vamos analisar um exemplo ilustrado na Figura 28. Os atores podem ser objetos, entidades externas aos sistemas ou subsistemas. Aqui o processo de entrega de uma encomenda é descrito. Um subsistema de controle de pesagem é ativado e o usuário é informado do custo e da previsão de entrega de sua solicitação.

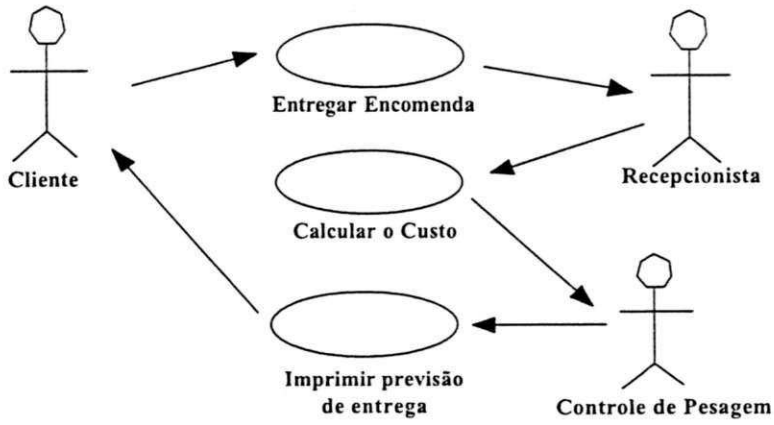

Figura 28: Diagrama de Caso de Uso da Entrega da Encomenda

A idéia é que cada caso de uso seja completo. Não é necessário que um único caso de uso cubra todo o sistema, nem que a abordagem de refinamentos sucessivos seja empregada. O objetivo é descobrir as reais necessidades do usuário através de exemplos pontuais simples e completos. É como se tirássemos fotos de

situações importantes dos procedimentos que fazem parte do domínio do problema.

O caso de uso da Figura 28 pode ter os fluxos de dados nomeados, como mostra a Figura 29 (fluxo como "Ordem de Viagem"). Entretanto, o foco central desta forma de capturar as funcionalidades do sistema e as necessidades do usuário reside na simplicidade dos casos de uso. Nesta fase, não devemos nos preocupar com o aspecto operacional de implementações ligadas à tecnologia usada.

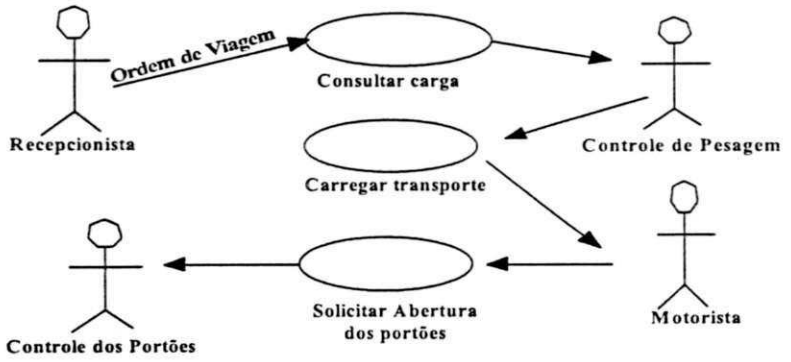

Figura 29: Diagrama de Caso de Uso do Despacho da Carga

2.5.3 Visão Lógica

A *visão lógica* utiliza as informações coletadas no caso de uso para construir o Diagrama de Classes. Os conceitos de objeto, classe, generalização e especialização (herança), e agregação são utilizados para modelar as informações importantes do domínio do problema que foi captado nos casos de uso.

2.5.3.1 Diagrama de Classes

O *diagrama de classes* apresenta as classes de objetos e seus relacionamentos. A representação de classes é feita através de retângulos, como mostra a Figura 30. As linhas que unem as classes significam os relacionamentos entre elas. Há uma indicação de uma cabeça de seta cheia (▸) para dizer o sentido da leitura do relacionamento. A cardinalidade do relacionamento é estabelecida pelo número de objetos de cada classe que participam da associação. Ela pode ser representada por um número *m*, por um asterisco *, por um intervalo *m..n*. Se *m* aparecer na especificação vai implicar a quantidade de objetos envolvidos no relacionamento. Já o asterisco indica "zero ou mais objetos". Quer dizer, o relacionamento pode ou não existir. A notação *m..n* é usada para indicar que o relacionamento deverá ter no mínimo *m* e no máximo *n* objetos.

A Figura 31 mostra os detalhes relativos às propriedades e comportamentos de uma classe. Os ambientes CASE como o Rational Rose (http://www.rational.com), o Together (http://www.together.com), SystemArchitet [MANS 90], ObjectMaker [MANO 90] e outros possibilitam automaticamente a geração da interface da classe em uma linguagem de programação orientada a objeto, baseada nos relacionamentos gráficos.

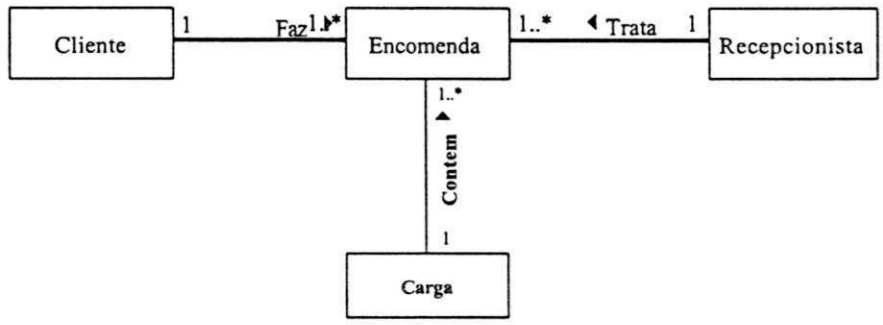

Figura 30: Exemplo de Classes e seus Relacionamentos

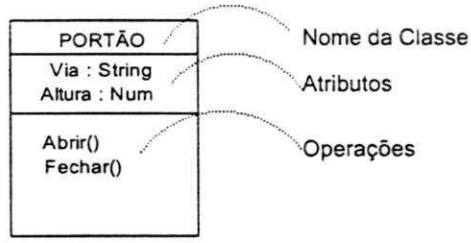

Figura 31: Características e Comportamento de uma Classe

2.5.3.1.1 Generalizações e Especializações

As informações e as operações ou comportamentos das classes mais gerais são herdadas pelas classes mais especializadas. A generalização, como pode ser vista na Figura 32, tem sua representação gráfica semelhante aos métodos anteriores. Acima, a classe mais genérica e abaixo, as classes mais especializadas que "herdam" características da classe superior. No exemplo, o fato de a classe *Carro* ser uma especialização de *Transporte* faz com que ela herde as características *Placa* e *Cor* e o comportamento *Matricular*.

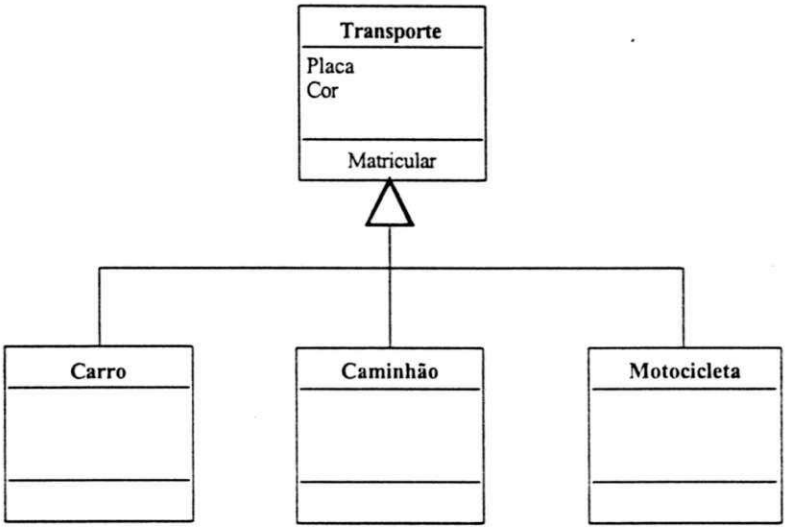

Figura 32: Exemplo de Generalização em UML

Em UML, entretanto, existem as generalizações restritivas. Podemos ter generalizações restritas de sobreposição, disjuntiva, completa e incompleta. Estas disciplinam a forma como a generalização será executada, como uma subclasse vai herdar características de múltiplas superclasses.

Vejamos um exemplo baseado em [UML 99]. Suponhamos que nosso transporte possa ser feito via barco e um tanque de guerra (anfíbio). Teremos que usar a restrição de *{sobreposição}* para indicar que as instâncias das classes *Carro* e *Barco* terão suas características sobrepostas. Já as instâncias das classes *Tanque* (anfíbio) herdarão características das classes *Barco* e *Carro* (vide Figura 33).

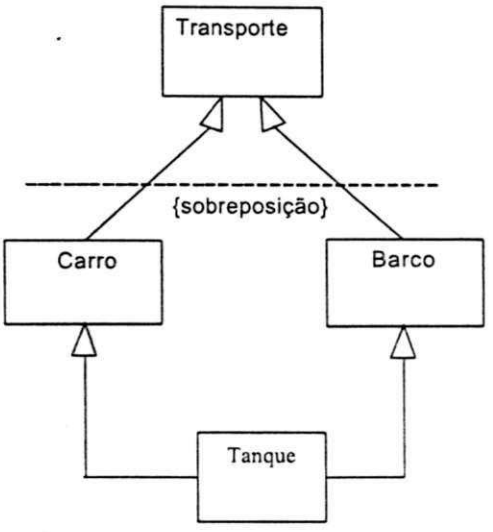

Figura 33: Herança Múltipla com Restrição de Sobreposição em UML

A *generalização completa* indica que todas as subclasses já foram especificadas sem a possibilidade de criação de outras especializações a partir daquele ponto. É exatamente o contrário da *generalização incompleta* Esta última é utilizada em especificações onde o domínio do problema exige que novas especializações sejam feitas. Por exemplo, podemos ter pessoa, como classe genérica e aluno e professor como suas especializações. Neste caso, nada impede que uma nova especialização, digamos secretária, seja feita de pessoa.

2.5.3.1.2 Agregação

A *agregação* é o relacionamento que envolve o todo e as partes. Uma das classes do relacionamento representa o todo e as demais as partes componentes. Existem a agregação compartilhada e a composta.

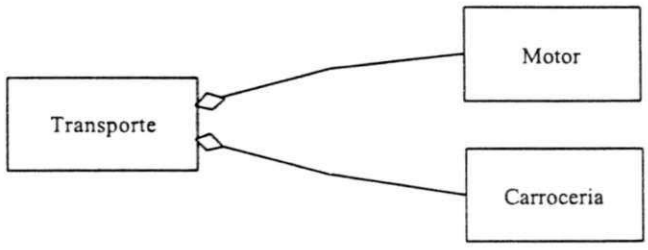

Figura 34: Exemplo de Agregação Composta

A Figura 34 ilustra que um transporte é composto de motor e carroceria. A representação agregação compartilhada usa o losango cheio, como mostra a Figura 35. Podemos recorrer ao conceito de passagem de parâmetros por valor e por referência, em linguagem de programação, para explicar a diferença com relação à agregação por composição. As instâncias das classes compartilhadas têm referências para as partes componentes, enquanto na composição, os valores dos componentes são partes integrantes do todo.

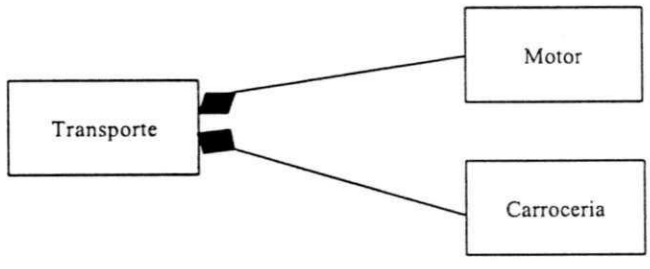

Figura 35: Agregação Compartilhada em UML

2.5.3.2 Diagrama de Objetos

A notação de objetos indica uma instância de um elemento que pertence a uma classe. Sua representação é similar à representação das classes, só que os nomes dos objetos devem ser sublinhados e separados por dois pontos do nome da classe que ele pertence,

conforme podemos ver na Figura 36. Supondo que exista uma classe *Cliente*, *Anna Carolina* representa uma instância da mesma.

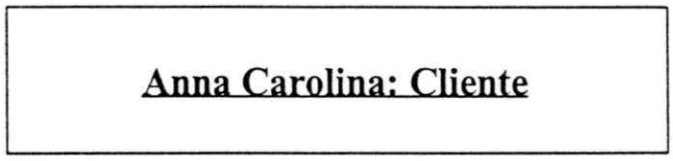

Figura 36: Representação de um Objeto em UML

A representação de objetos é útil em outros diagramas de UML que precisam modelar outros aspectos envolvendo objetos. Podemos citar o diagrama de seqüência, colaboração e de atividades que veremos adiante.

Podemos pensar no diagrama de objetos ainda como uma forma de ajudar a esclarecer situações onde fica difícil encontrar o Diagrama de Classes para o domínio de um problema mais complexo.

2.5.4 Visão Dinâmica ou Concorrente

A Visão Dinâmica apresenta os diagramas de Estado, Seqüência e Atividades, para mostrar aspectos dinâmicos de uma aplicação. Estes recursos são utilizados também para documentar restrições impostas à construção de sistemas baseadas em características dinâmicas do escopo do problema ou da solução tal como desempenho, por exemplo.

2.5.4.1 Diagrama de Estado

Os *estados* são representados por retângulos com cantos arredondados, como mostra a Figura 37. Existem dois estados

especiais rotulados de início e fim. As setas no diagrama indicam o sentido e o evento (o rótulo) que proporcionou a mudança de estado. Existem outras formas mais elaboradas, que não abordaremos aqui, para lidar com temporizadores e eventos condicionais.

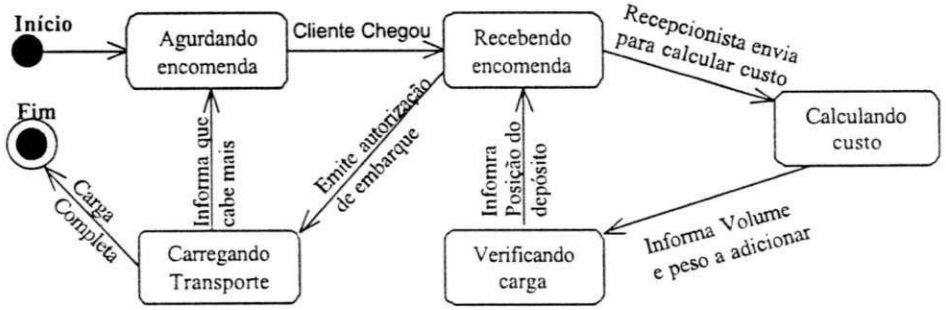

Figura 37: Diagrama de Estados da Recepção em UML

No exemplo da Figura 37, o cliente chega e entrega uma encomenda a um recepcionista. Este, após cálculo do custo e verificação, emite uma ordem de carga. Este diagrama dá uma visão geral do processo. Vejamos o diagrama de Seqüência, a seguir, para ver outros aspectos dinâmicos do sistema.

2.5.4.2 Diagrama de Seqüência

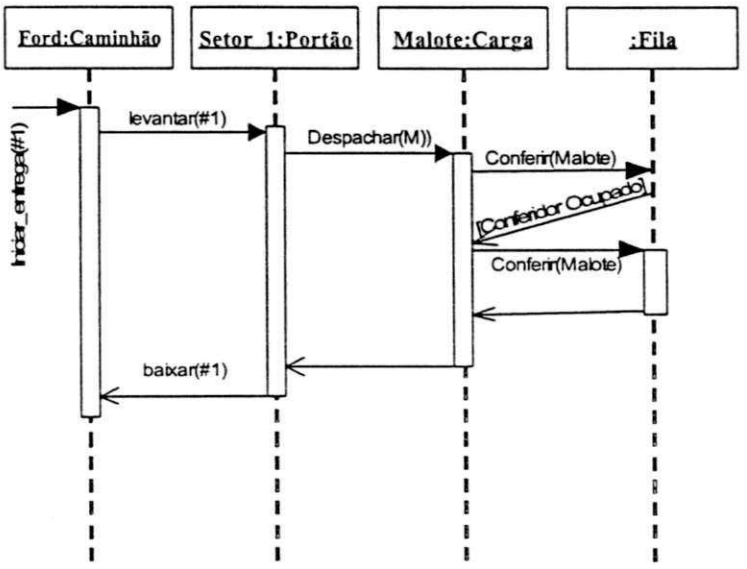

Figura 38: Diagrama de Seqüência da Entrega de um Malote em UML

O *Diagrama de Seqüência* mostra a colaboração dinâmica entre os vários objetos do sistema. É possível visualizar a seqüência de mensagens e o ciclo-de-vida dos objetos durante uma seqüência de execução. Um exemplo é mostrado na Figura 38. Podemos observar a seqüência envolvendo os objetos *Ford*, *Setor_1*, *Malote*, instâncias, respectivamente, das classes *Caminhão*, *Portão* e *Carga*. Existe ainda um terceiro objeto, sem nome, que pertence a classe *Fila*. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam. Podemos visualizar dois eixos. O vertical, que mostra o tempo e o eixo horizontal, que mostra os objetos envolvidos na seqüência para liberar uma carga.

No eixo horizontal estão os objetos envolvidos na seqüência. Cada um é representado por um retângulo de objeto e uma linha vertical pontilhada chamada de linha de vida do objeto, indicando a execução do objeto durante a seqüência. A comunicação entre os objetos é representada como linha com setas horizontais simbolizando as mensagens entre as linhas de vida dos objetos.

Observe que uma restrição no Diagrama de Seqüência pode indicar o tempo máximo que um caminhão pode levar para despachar um malote. Este diagrama pode fornecer informações valiosas a serem usadas na escolha de tecnologias que implementarão a funcionalidade *despachar* um *malote*.

2.5.4.3 Diagrama de Colaboração

Assim como o diagrama de seqüência, o *diagrama de colaboração* mostra a colaboração entre objetos para executar uma atividade. Note no exemplo da Figura 39, que equivale ao diagrama da Figura 38, que o tempo não aparece. O Diagrama de Colaboração é mais indicado para analisar o contexto geral dos objetos envolvidos. A numeração das mensagens serve para indicar a ordem de ativação.

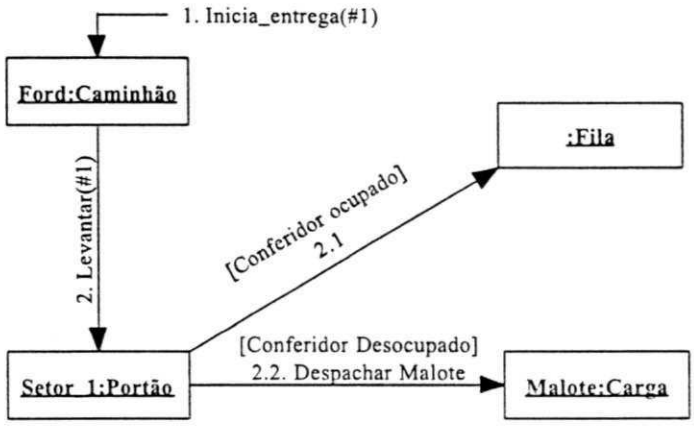

Figura 39: Diagrama de Colaboração em UML

2.5.4.4 Diagrama de Atividade

O *Diagrama de Atividade* serve para analisar o comportamento das instâncias de alguns métodos dos objetos de um grupo de classes. Assim como o Diagrama de Estado, ele possui um estado inicial e um final, como pode ser visto no exemplo da Figura 40. Note que o losango insere uma bifurcação ou ponto de decisão, indicando qual o fluxo a seguir dependendo se o *Conferente* estiver ocupado ou não. Ele também pode ser usado para descrever um fluxo de eventos em um caso de uso.

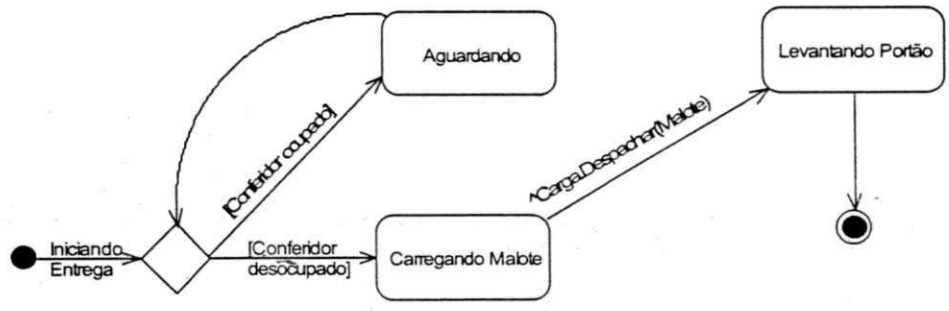

Figura 40: Diagrama de Atividade de Despachar Malote em UML

2.5.5 Visão de Componentes

O Método UML disponibiliza dois diagramas com o objetivo de modelar componentes presentes no domínio do problema a ser especificado para posterior construção do sistema. São eles o Diagrama de Componentes e o Diagrama de Implementação. Ambos estão associados às fases mais próximas da implementação.

2.5.5.1 Diagrama de Componente

Um componente é qualquer parte do sistema que está sendo construído. Sua representação encontra-se na Figura 41. Seu objetivo é analisar os relacionamentos entre módulos ou artefatos de software, através de dependências. A Figura 42 ilustra como funciona o comando "*make*" que ficou famoso por aumentar a produtividade dos programadores no ambiente de desenvolvimento de software, isto porque ele automatizou as atividades de compilação e outras atividades que antes eram feitas manualmente [MEDE 94].

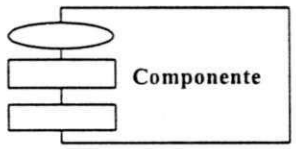

Figura 41: Representação de um Componente em UML

O mecanismo de funcionamento do comando *make* é simples: existe um arquivo chamado "*makefile*" que implementa a dependência descrita na Figura 42. O programador simplesmente ativa o comando *make* e ele descobre o que fazer para gerar o executável *prog.exe*. Como as setas pontilhadas na figura ilustram, há dependência entre este e os arquivos terminados em "*.obj*". Assim, o compilador é ativado automaticamente para compilar apenas os arquivos fontes

com terminação ".c" que foram modificados e gerar as respectivas versões executáveis.

Componentes podem definir interfaces que são visíveis para outros componentes.

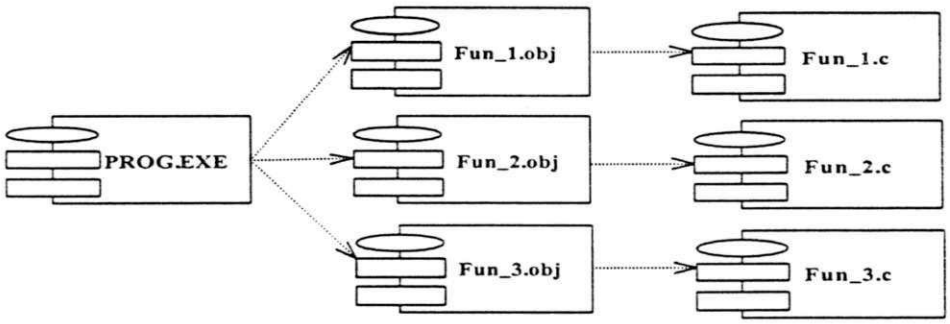

Figura 42: Árvore de Dependência do *make* usando Componentes em UML

2.5.5.2 Diagrama de Implementação

A preocupação central do *diagrama de implementação* é fornecer construções para modelar aspectos físicos do sistema em construção. Observe na Figura 43 que detalhes de implementação, como a configuração de hardware e software, os meios e a tecnologia de comunicação, são modelados para o controle de entrega de encomendas.

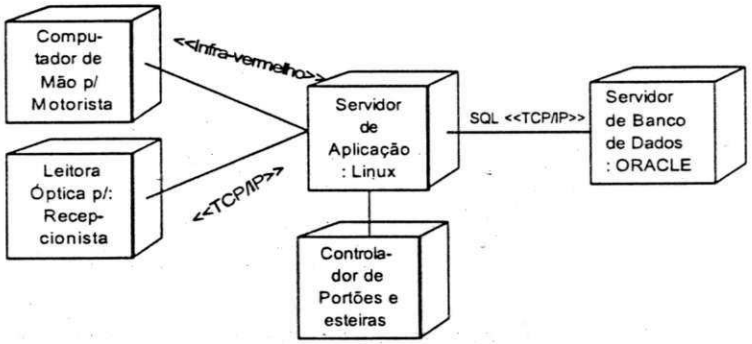

Figura 43: Exemplo de Diagrama de Implementação em UML

2.6 Classificação dos Conceitos

Apresentamos uma visão geral de cada MESOO. Aqui estamos interessados em verificar como os principais conceitos dos MESOOs podem ser agrupados para eliminar a contextualização dos termos empregados nas suas respectivas fontes bibliográficas ([UML 97/98], [COAD 90]/ [COAD 91], [SHLA 88]/ [SHLA 91], e [WIRF 90]).

As metodologias associadas aos MESOOs descrevem os processos necessários para a utilização dos seus modelos ou submodelos. Como podemos ver em Sanders [SAND 95] "*existem muitos MESOOs e ferramentas para o desenvolvimento de aplicações e pouca pesquisa é dedicada ao entendimento sobre quais ferramentas facilitam o processo de desenvolvimento de software.*" Muitos analistas de sistemas acreditam que a experiência está baseada no conhecimento de princípios gerais e que estes podem ter impacto positivo no processo de desenvolvimento de software.

Apesar de não ser o foco central de nosso trabalho, a identificação e classificação dos conceitos fundamentais dos MESOOs gera uma perspectiva crítica estabelecendo parâmetros que favorecem o aprendizado ou a análise de novos MESOOs. O centro de nosso trabalho está na proposição de um metamodelo para STL, onde os conceitos dos MESOOs e os conceitos da STL possam ser mapeados.

A busca de uma notação comum reduz o efeito contextual dos termos de cada MESOO e contribui para a formação do

metamodelo. Assim, destacamos a seguir algumas comparações entre metodologias de especificação de software encontradas na literatura.

2.6.1 Comparações entre Metodologias

Existem algumas estudos que comparam metodologias de especificação de software. Elas definem uma forma de classificar os conceitos das metodologias e fazem uma análise das ferramentas de modelagem conceitual suportadas pelas metodologias. Vejamos o resumo das principais fontes encontradas na literatura. Nosso objetivo é encontrar uma notação comum que reduza o efeito contextual dos termos de cada MESOO.

Kucera [KUCE 94] fez uma comparação da notação gráfica e relacionou com os diagramas presentes nos métodos de cada Metodologia de Especificação de Software (MES) estudada. Iivari [IIVA 95] classificou seis MES segundo os aspectos funcionais, estruturais e comportamentais. Ekert e Golder [EKER 94] compararam quatro métodos focalizando apenas os aspectos estruturais. Mitchel Lubar *et al.* [LUBA 94] utilizaram um exemplo em comum para verificar os pontos fortes e fracos dos MESOOs, utilizados ao longo da modelagem de um problema. Na realidade, não é relevante para nosso trabalho esta classificação dos conceitos fundamentais em grupos. Estamos interessados em trabalhar com os conceitos dos MESOOs, em nível de granularidade que permita sua representação em STL de forma canônica.

Apresentamos os conceitos das metodologias que acompanham os MESOOs segundo seus aspectos estruturais, contextuais e comportamentais. Esta classificação está muito próxima

da classificação de Martin Fowler [FOWL 95]. A diferença está em como classificar os componentes arquiteturais do sistema. Em nossa visão, eles dependem do contexto de cada metodologia devido ao vocabulário próprio de cada um dos métodos estudados. Cabe ao desenvolvedor identificar, na bibliografia disponível sobre cada MESOO, o que é mais relevante levando em consideração o domínio do problema a ser modelado. Apresentamos a seguir três tabelas a título de exemplo de como os conceitos dos MESOOs poderiam ser classificados. Estas informações serão essenciais para uma melhor compreensão do potencial semântico de cada um dos MESOOs estudados e importantes para a formação de um senso crítico para analisar novos MESOOs.

Tabela 2 : Aspectos Estruturais

Conceito	AOO	UML	SHAM	WOOD
Representação Objeto e classe	Objeto e Classes	Objeto e Classe	Classe	Classe e objetos[12]
Generalização/ Especialização	"is a", "is kind of"	Herança "is a" "has"	Supertype/ Subtype "is a"	"is kind of"
Classe abstrata	Suportada	Suportada	Não	Suportada
Agregação	Estrutura "whole-part"	Composta e compartilhada	"consist of"	"is part of"
Associação	Objetos associativos	"role names" "uses" Associação	Objetos associativos	Colaboração
Condições na associação	Sim	Opcional com a cardinalidade	Especificado separadamente	Com multiplicidade
Multiplicidade da associação	Sim	Sim	Sim	Não
Atributos nas associações	Tratados como objetos normais	Atributos de ligação	Objetos associativos	Não
Definição dos serviços	Texto, cartazes	Casos de uso	DFD	Protocolos
Herança múltipla	Sim	Sim	Sim	Sim

[12] Utiliza uma notação para os dois conceitos

A Tabela 2 mostra os aspectos estruturais dos modelos de especificação de software orientados a objetos. Observe que todas as metodologias têm alguma forma de lidar com objetos, classes, herança (uma forma de generalização e especialização de classes), agregação, e associação. Nos textos base das metodologias, às vezes, os autores referem-se a estes conceitos com diferentes nomes, mas com o mesmo significado. É importante que o desenvolvedor visualize estes componentes no jargão contextual de cada metodologia para entender sua extensão. Podemos observar que, com exceção do método de Shlaer [SHLA 88, 91], todos os métodos têm representações gráficas distintas para objetos e classes. Podemos ver que o Método AOO [COAD 90, 91] apesar de ter uma representação específica para classes abstratas, sobrecarrega um único símbolo gráfico com dois significados: representar classes concretas e objetos.

O conceito de herança (generalização/especialização) está presente em todas as metodologias estudadas. Os autores em seus textos referem-se a este conceito dizendo que um objeto "é um", "é um tipo de", ou falam de subtipo e supertipo. O único texto que não trata de classe abstrata é o de SHAM [SHLA 88, 91].

À agregação é dada uma atenção especial em AOO nas estruturas "todo-parte". Os demais métodos usam expressões do tipo "é parte de" e "consiste de".

O conceito de associação existe em todos os métodos de especificação de software orientados a objetos. Há uma riqueza de termos e estes devem ser interpretados com base no contexto do jargão de cada metodologia. Podemos falar em termos de objetos que

colaboram, ou classes de objetos cujos relacionamentos podem ser descritos por meio de objetos. O nível de detalhamento ou refinamento desejado vai determinar se as condições na associação devem ser explicitadas. Outro fator que é levado em consideração é a multiplicidade e os atributos da associação. O único método que não especifica estes aspectos é WOOD [WIRF 90]. Coad *et al* [COAD 91] tratam os atributos das associações como objetos normais; UML [UML 96] trata como classes ou atributos de ligação; Shlaer *et al* [SHLA 88, 91] referem-se a objetos associativos.

A definição do comportamento, das operações ou serviços pode ser feita através de texto; cartazes; casos de uso; diagrama de fluxo de dados ou protocolos (responsabilidades).

Todas as metodologias tratam o sistema com diferentes medidas de unidades. Neste caso, os conceitos são agrupados para minimizar a complexidade do sistema. Podemos ver que existem diferentes representações para a realidade em cada uma das metodologias e que o peso para cada um dos aspectos estrutural, contextual e comportamental varia em cada modelo, dependendo do nível de profundidade e detalhamento das referências bibliográficas.

As ferramentas de modelagem conceitual definidas em cada MESOO determinam quais modelos melhor se aplicam a um problema ou domínio específico. AOO e SHAM são indicados para modelar sistemas de informação, enquanto WOOD é indicado para controle de processos ou controlar o comportamento de um sistema. UML é o mais rico de todos e tem construções semânticas que cobrem os dois escopos acima citados.

Todos os métodos apresentam uma ou mais formas de representação de estados. Podemos ter um grafo de colaboração, ou diagramas de estados.

O subjetivismo das classificações nos leva a refletir sobre a dificuldade na identificação precisa dos conceitos envolvidos nos MESOOs. Entretanto, a visão genérica dos MESOOs seria justificada para uma abordagem onde as ferramentas CASE teriam que, num futuro próximo, compartilhar um padrão de representação comum para trocar informações em diferentes níveis de granularidade de conceitos, sem, necessariamente, ter seus fontes expostos à concorrência. Neste sentido, as primitivas de STL podem ser utilizadas para representar os conceitos dos MESOOs.

2.7 Cenário Escolhido

Como vimos nas seções anteriores, cada MESOO conta com um jargão de termos utilizados para descrever os conceitos e construir a especificação. Sabemos que as construções usadas em algum MESOO podem não encontrar mecanismo correspondente em outro método de especificação. Assim, propomos um cenário restrito às representações de herança, agregação e uma associação:

Um transporte ou é um carro, ou uma moto, ou um caminhão;

Cada transporte está associado a um portão;

Uma carga consiste de encomendas e um roteiro de entregas.

As representações gráficas dos cenários serão discutidas no próximo capítulo, quando apresentaremos as dificuldades de transferir semântica no metamodelo atual de STL.

Apresentamos no próximo capítulo uma visão geral do metamodelo atual de STL e apresentaremos suas limitações. Situaremos STL no contexto do Padrão IEEE 1175.

Capítulo

3

Visão Geral de STL e Suas Limitações

Apresentamos neste capítulo uma visão geral de STL. Descreveremos os principais conceitos e mecanismos envolvidos para a transferência de informações no Padrão IEEE 1175 [IEEE 92-99].

Discutiremos — com o cenário do capítulo anterior que ilustra o uso dos conceitos de herança, associação e agregação dos MESOOs de interesse — as deficiências do metamodelo STL, de modo a contextualizar a proposta de um novo metamodelo que enderece as deficiências ilustradas.

3.1 Objetivo de STL

O objetivo de STL é definir mecanismos para transferência de informações entre ferramentas CASE na organização, entre plataformas de computadores, e entre aplicações de software.

Vejamos a seguir o modelo de referência do padrão IEEE 1175, através da Figura 44, que atende ao propósito de ilustrar o mecanismo geral de funcionamento do Padrão IEEE 1175 e qual o papel da Linguagem STL. Mais detalhes podem ser encontrados no documento IEEE 1175 [IEEE 92-99]. Observe que a necessidade de comunicação pode ocorrer em vários níveis ou contextos: organizacional, arquitetural e entre ferramentas CASE.

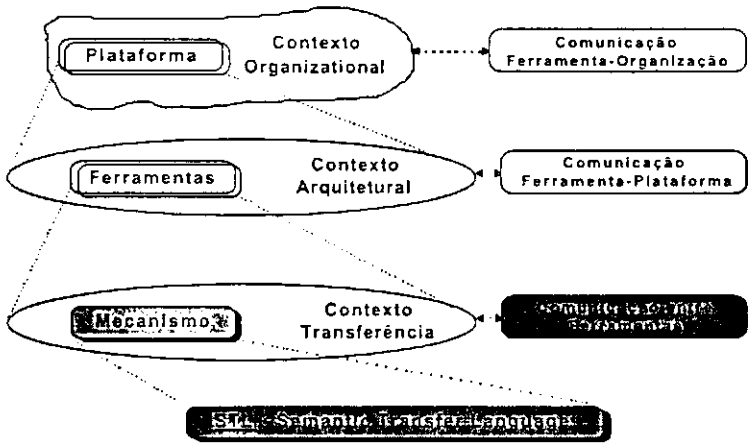

Figura 44: Modelo de Referência 1175

O contexto organizacional estabelece mecanismos para se adequar ou integrar uma ferramenta a um padrão previamente estabelecido na organização. Já o contexto arquitetural está voltado para a definição de como os componentes físicos da ferramenta devam se comportar em grupos ou famílias com o objetivo de promover uma melhor integração do ambiente operacional. Estes dois contextos não são relevantes para a discussão do metamodelo STL e não serão discutidos em detalhes nesta tese.

O contexto da transferência é onde a Linguagem de Transferência de Semântica (STL) é usada. As informações a serem transferidas entre ferramentas CASE são empacotadas. Cada pacote é

descrito com base em um catálogo e no seu próprio conteúdo, conforme veremos a seguir.

3.2 Descrição em STL

Uma descrição STL é composta de duas partes: a descrição ou catálogo da informação a ser transferida (o formato) e o pacote ou conteúdo propriamente dito a ser transmitido, conforme podemos ver na Figura 45.

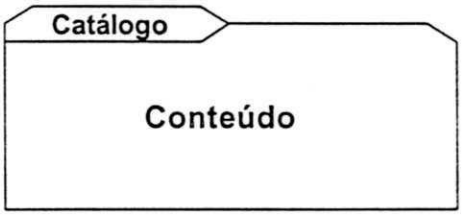

Figura 45: Formato de uma Descrição em STL

As informações a serem transferidas utilizam uma notação gráfica (os diagramas nos MESOOs) para expressar a especificação de uma aplicação. A forma (sintaxe) e o significado (semântica) dos diagramas precisam ser transmitidos entre as ferramentas CASE FC_a e FC_b para que a comunicação seja executada com sucesso. Suponhamos que, a título de ilustração para construção de uma descrição STL, algum MESOO m possua o diagrama como mostra a Figura 46.

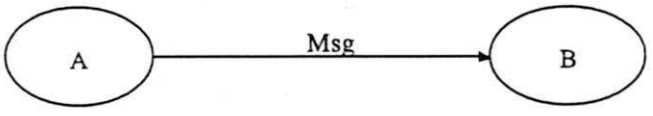

Figura 46: Exemplo de Comunicação de Processos em m

Suponhamos ainda que estas informações gráficas precisam ser transmitidas de uma ferramenta CASE FC_a para outra FC_b. Um pacote STL deve ser construído contendo o mapeamento dos conceitos da Figura 46 para uma representação STL textual. Neste formato, o pacote é transmitido de FC_a para FC_b e remontado no destino. O catálogo está relacionado com a identificação da informação a ser transferida.

3.3 Identificação da Informação a ser Transferida

O papel do catálogo nos pacotes STL é o de identificar a informação a ser transferida entre duas ferramentas CASE. Por exemplo, observe na Figura 47, a informação de que o pacote a ser transmitido é composto de dois processos (A e B) e um fluxo de informação (*Msg*) precisa ser enviada à ferramenta que vai receber o pacote.

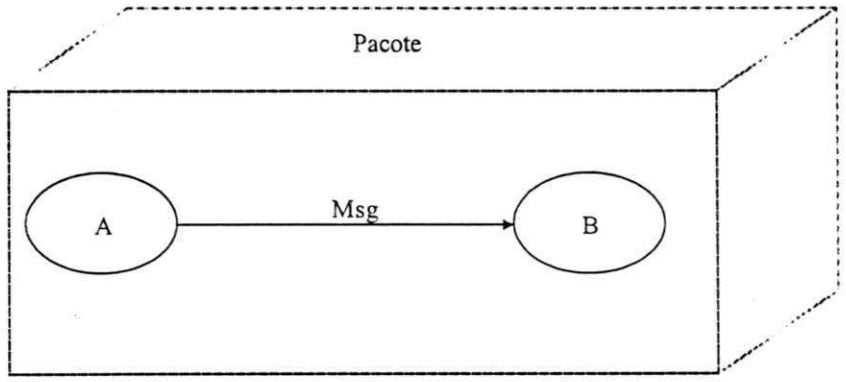

Figura 47: Pacote contendo diagrama

Obviamente, esta informação precisa chegar antes da transmissão do pacote propriamente dito para que a ferramenta receptora possa utilizar a semântica transmitida. Resumindo, é

necessário que se estabeleça um quadro comum de comunicação entre as ferramentas. No exemplo podemos usar *Diagram_AB*, descrito em STL a seguir, como sendo a informação principal a ser transmitida. Observe que há uma composição de dois processos e um fluxo de informação. Necessariamente, a descrição precisaria ser catalogada na ferramenta receptora da seguinte maneira:

```
1 diagram Collection        Diagram_AB;
2 process Actions           Proc_A,
                            Proc_B;
1 dataflow ConnectionPath   Flow_AC for "Msg" into Proc_B;
```

A descrição acima faz com que a ferramenta receptora catalogue o *Diagram_AB*. Em seguida, podemos descrever ou transmitir o pacote de informações propriamente dito (*T_Packet*).

3.4 O Pacote de Informações

Como pode ser visto em mais detalhes no Apêndice A, STL utiliza uma notação que é uma extensão da sintaxe BNF ("*Backus-Naur Form*") [NAUR 60]. Os conceitos são descritos de forma *top-down*. Observe que um pacote tem o início e fim delimitados pelas palavras-chave *T_Packet* e *pe_mark*, respectivamente:

```
T_Packet                Diagram_AB;
has subject             "Comunicação entre A e B";
has content version     "Versão 1";
has creator             "Álvaro F. C. Medeiros";
has description         "Pequeno exemplo desc. STL";

Collection              Diagram_AB
has label               "Diagrama A -> B";
groups action           Proc_A,
```

```
                                        Proc_B;
groups connectionpath                   Flow_AB;

Action                                  Proc_A
has label                               "A";
is actiontype                           internal;
has transform purpose                           data;
is connected to connectionpath          Flow_AB;

Action                                          Proc_B
has label                               "B";
is actiontype                           internal;
has transform purpose                           data;
is connected from connectionpath        Flow_AB;

ConnectionPath                                  Flow_AB
has label                               "Msg";
is connectiontype                       data;
connects from action                    Proc_A;
connects to action                      Proc_B;
is grouped into collection              Diagram_AB;
pe_mark         /* end of the package */
```

Note que os pacotes são decompostos em cláusulas e sentenças. Estas, por sua vez, podem ser diretamente interpretadas pelo computador e são formadas de uma palavra-chave seguida por um identificador de cláusula. Note que as palavras-chave vêm do metamodelo de STL (data, action, event,...) e de suas cláusulas (veja resumo na Tabela 5).

Tabela 5 : Palavras-chave das Sentenças em STL

Action	DataPart	EventType
Collection	DataRole	GraphicalSymbol
Condition	DataStore	Object
ConnectionPath	DataType	T_Packet
Constant	DataView	State
DataItem	EventItem	StateTransition
DataKey		

Cada palavra-chave refere-se a um conceito a ser usado na descrição do escopo do sistema sendo especificado, podendo existir

uma série de cláusulas complementares obrigatórias e/ou opcionais. As cláusulas são de dois tipos: aquelas que descrevem relacionamento entre conceitos, e aquelas que descrevem atributos de um conceito. Mais adiante vamos trabalhar com os conceitos de STL e seus relacionamentos.

Como vimos, o IEEE 1175 [IEEE 92-99] foi projetado para estabelecer um padrão de transferência de informações em toda a organização. A Linguagem STL está inserida no contexto operacional para permitir que as ferramentas CASE se comuniquem. STL preocupa-se com o contexto de transferência. STL é um esforço para definir uma forma comum de comunicação entre ferramentas distintas. Entretanto, o propósito do padrão como um todo é estabelecer mecanismos para a transferência de informação nos outros contextos organizacionais e arquiteturais. O modelo de referência para o contexto de transferência está dividido nas seguintes partes [IEEE 92-99]:

- Mecanismo de comunicação;
- Processo de transferência de informações;
- Descrições das informações transferidas.

3.4.1 Mecanismo de Comunicação

O mecanismo para transferência de informações pode utilizar uma combinação de hardware e software, como podemos ver na Figura 48, extraída de IEEE 1175 [IEEE 92-99], podendo envolver diferentes mecanismos de comunicação (direto, baseado em arquivos, baseado em um repositório central, ou baseado em protocolos e redes).

3.4.2 Processo de Transferência de Informações

Do ponto de vista de STL, o mecanismo de comunicação é transparente. STL não se preocupa com detalhes físicos de comunicação, se a informação vem de um arquivo temporário ou de uma linha de comunicação. A Figura 49 ilustra como uma ferramenta A pode se comunicar com outra B usando apenas duas primitivas send e receive, independente do mecanismo de comunicação.

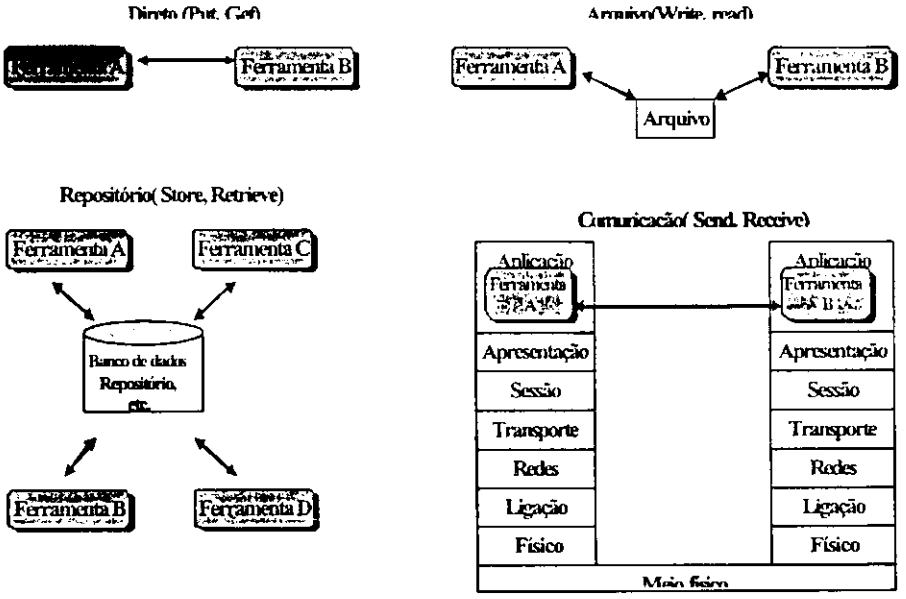

Figura 48: Exemplos de Mecanismos de Comunicação

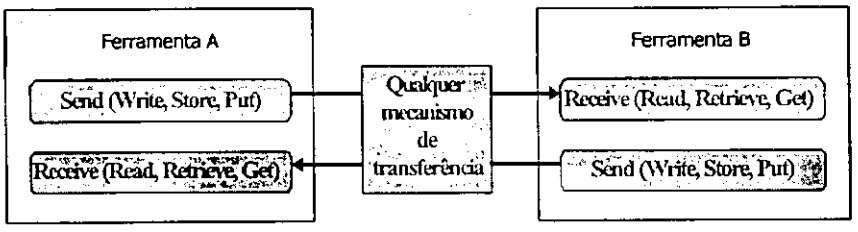

Figura 49: Interconexões entre Processos

3.4.3 Descrição da Informação Transferida

É necessário descrever a informação a ser transferida entre diferentes ferramentas. Isto deve ser feito para que ambas as ferramentas compartilhem a mesma interpretação da informação. Desta forma, cada unidade de informação transferida deve ter sua sintaxe (forma) e sua semântica (significado) definidas.

3.4.4 Informação Transferida

A informação transferida pode ser classificada como de controle, de gerenciamento do processo de comunicação, ou o conteúdo de um assunto específico de uma aplicação. As informações de controle permitem as ferramentas iniciar, parar, enviar, receber, e reenviar informações. As informações de gerenciamento servem para que o interpretador possa montar o catálogo e checar o conteúdo ou assunto sendo transferido. Cada informação sobre conceitos precisa de gerenciamento e controle. O conteúdo da informação ou assunto inclui requisitos de software, projetos, programas, e testes. Os assuntos são descritos em termos dos conceitos base do metamodelo STL e seus relacionamentos.

3.5 O Metamodelo de STL

O metamodelo existente de STL descreve os elementos (*dado, ação, lógica (condição, restrição), evento*, e *estado*) que fundamentam a troca de semântica entre as ferramentas CASE. Como podemos ver na Figura 50, as representações dos MESOOs são sucessivamente mapeadas em visões até serem reduzidas aos conceitos de STL. É o catálogo do pacote a ser transferido que vai garantir que as

ferramentas CASE estarão utilizando a mesma semântica das informações transferidas.

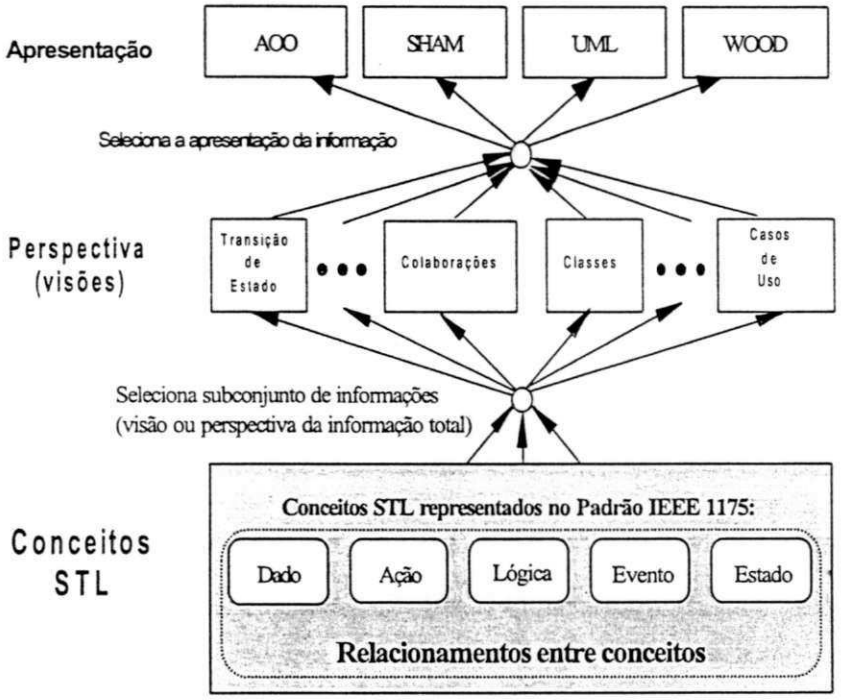

Figura 50: Visão geral do Metamodelo de STL

O conceito *ação* pode representar comportamento, controle, e transformação de *estados*. O conceito *dado* representa informação. O conceito *evento* está associado a ocorrência de ações em um determinado tempo. O conceito *lógica* relaciona *condição* ou *restrição* a *ações* e *eventos*.

3.6 Implementação de um Interpretador de Pacotes STL

O metamodelo de STL resume-se a um documento que estabelece como as ferramentas CASE, de diferentes distribuidores,

devem se comportar para implementar um mecanismo de comunicação. [IEEE 92-99]

Nossa proposta de criação de um interpretador de pacotes (*IP*) visa automatizar a interpretação de especificações escritas em STL e fornecer uma biblioteca de funções que possa ser oferecida às ferramentas CASE que desejarem se comunicar, sem a necessidade de conhecimento prévio da estrutura interna de cada uma das ferramentas. Podemos imaginar a computação ou simulação de uma configuração executável *CE* feita através do reconhecimento pelo *lex* e *yacc* [SHAM 99, GORM 96] das cláusulas STL.

A construção do *IP* foi executada em função de dois utilitários que constroem, respectivamente, um analisador léxico e um analisador sintático, (vide mais detalhes da implementação no Anexo A pp. 151) a partir de adaptações para colocar o IEEE 1175 no formato requerido pelo *lex* e *yacc*.

Os diagramas STL são transformados em texto, através de seus conceitos básicos e derivados, na ferramenta de origem e são empacotados e enviados para a ferramenta de destino. Lá o pacote é remontado e o diagrama original é recomposto. Assim, é possível visualizar as ações que estão sendo feitas através do reconhecimento dos conceitos de STL.

3.7 Limitações de STL

A limitação de STL está ligada a seu objetivo inicial que é o de oferecer conceitos que permitam as ferrementas se comunicarem através de troca de pacotes de informações. Como podemos ver na

sua especificação [IEEE 92-99], os conceitos STL estão voltados para a transformação de representações gráficas (os diagramas dos MESOOs) em texto e a especificação de como montar os pacotes a serem transferidos.

Podemos constatar esta limitação, através da execução do Interpretador de Pacotes *IP*

$$\$\, ip < E_{Am}$$

que interpreta uma especificação E_{Am} lendo seu conteúdo e, a partir dele, monta seu catálogo. Isto demostra que não existem cláusulas nos pacotes STL que permitam que conceitos usados em uma especificação de uma aplicação E_{Am} (no seu formato textual no exemplo da linha de comandos do *Linux* acima) de um MESOO $m \in M$ seja comparado com outra especificação anteriormente transferida. Podemos observar através da saída de *ip* (através da listagem do catálogo correspondente ao conteúdo do pacote) que os pacotes construídos por STL não oferecem recursos para fazer equivalência entre conceitos sendo transmitidos em E_{Am} nem entre dois MESOOs diferentes.

O comando *ip* monta pacotes E_{Am} no formato textual de STL na ferramenta CASE fonte e os desmonta no destino.

No próximo capítulo, vamos propor uma representação gráfica para STL com o objetivo de visualizar o mapeamento entre os conceitos dos MESOOs e STL.

Capítulo

4

Proposta de Uma Representação Gráfica para STL

Apresentamos neste capítulo uma forma de representação gráfica para STL, com o objetivo de visualizarmos a equivalência entre conceitos dos MESOOs ao serem mapeados em STL.

4.1 Proposta de uma Notação Gráfica para STL

A descrição do metamodelo STL [IEEE 92-99] é feita em uma notação visando a criação de gramáticas em BNF[13]. Esta abordagem leva à automatização de pacotes entre as ferramentas CASE. O reconhecimento do metamodelo STL pode ser feito através de ambientes que utilizem o *lex* e o *yacc* para gerar e reconhecer pacotes escritos através da gramática STL [MEDE 2000]. Os

[13] Na realidade, o STL precisou ser adaptada para a Bakus Nau Form exigida pelos comandos *lex* e *yacc* na implementação do Interpretador de Pacotes *IP*.

MESOOs são descritos através da população de cláusulas do metamodelo STL em pacotes de informação.

Analisar quais cláusulas foram usadas para transferir semântica entre os MESOOs e poder visualizar as diferenças e sobreposições entre os métodos de especificação de software é uma atividade que demanda conhecimento mais detalhado de STL. Por outro lado, o metamodelo STL pode ser abordado ainda informalmente com a metáfora de banco de dados, como apresentado em Medeiros *et al* [MEDE 96, 98]. Os conceitos representam o esquema e a população representa a especificação. Avisão dos arquivos produzidos pelo mapeamento dos conceitos dos MESOOs para as cláusulas STL podem ser vistos como a população do esquema do metamodelo STL.

Outra possibilidade para formalizar o metamodelo acima é criar uma representação canônica cujos conceitos fiquem mais próximos dos conceitos utilizados pelos MESOOs. Esta forma foi adotada nesta tese para ilustrar de forma gráfica as sobreposições e diferenças entre especificações construídas por diferentes MESOOs.

4.1.1 Representação Gráfica do Metamodelo STL

A representação gráfica do metamodelo STL (RGM) se propõe a explicitar o mapeamento e o agrupamento de conceitos dos MESOOs durante a montagem e desmontagem de pacotes STL.

Escolhemos trabalhar com a RGM para poder agrupar conceitos do Metamodelo STL em um nível de granularidade mais elevado para melhor entender as particularidades de cada MESOO

estudado nesta tese e ilustrar o mecanismo de mapeamento para conceitos STL.

Usamos este artifício porque STL foi desenvolvida baseada em notação próxima de BNF. Lembramos que ambientes de geração de gramáticas, com o emprego de utilitários como o *lex* e o *yacc* poderão ser usados para trabalhar com os conceitos em seu nível mais detalhado através da implementação de gramáticas que reflitam os conceitos da Figura 50 (vide página 85)e seus relacionamentos. O metamodelo será descrito em termos de conceitos base e derivados.

4.1.1.1 Conceitos Base

Agruparemos os **conceitos base** de STL para reproduzir as entidades, processos, mensagens, a lógica (condições, restrições), os estados (valores) e suas transições. Descreveremos na seção seguinte os **conceitos derivados** como formas de combinar os conceitos base e formar elementos compostos para representar os MESOOs. Não se trata de definir mais um MESOO, apenas estamos buscando uma representação onde seja possível utilizar STL como base para representar MESOOs e observar suas características.

Entidades

Representaremos **entidades** do domínio do problema com a notação gráfica de um retângulo, como podemos ver na Figura 51. Esta é uma representação para as informações ou dados relevantes do domínio do problema. No nosso cenário, podemos dizer que Portão, Motorista e Caminhão são entidades que merecem destaque.

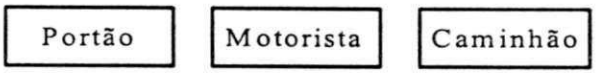

Figura 51: Representação de Entidades em STL

Os Atributos ou as características das entidades serão representados posteriormente, através de uma elipse, quando mais detalhes surgirem no cenário. Um exemplo pode ser visto na Figura 52.

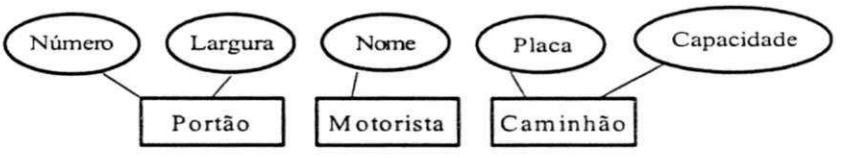

Figura 52: Entidades com Atributos

Processos

Observamos que as *ações* STL não existem isoladamente. Elas são agrupadas em **processos** e estão relacionadas às atividades executadas no domínio do problema. Podemos representar os processos através de retângulos com cantos arredondados, como ilustra a Figura 53. Podemos ilustrar o processo de liberação do transporte para iniciar a entrega através do processo *levantar portão.*

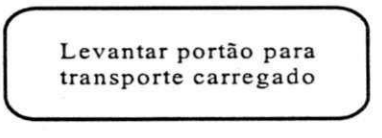

Figura 53: Representação de Processo em STL

Mensagens

As *entidades* relacionam-se com os processos através de *mensagens* Na realidade, as *mensagens* vão interagir com os *eventos* STL para estimular as *ações* (processos). Podemos ver a representação gráfica de uma mensagem na Figura 54. Uma mensagem representa a comunicação entre entidades ou processos do mundo real. É através de mensagens que informamos a ocorrência de eventos.

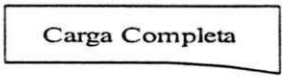

Figura 54: Representação de Mensagem em STL

Condição (Lógica)

O conceito de lógica do metamodelo STL pode ser uma condição ou uma restrição podendo ter o valor de verdadeiro ou falso. Uma expressão é avaliada com base em seus operadores lógicos e operandos (*1,2,...n*). Os operandos são representados graficamente pelo conceito derivado *operando*, descrito na Seção 4.1.1.2

A título de exemplo, representamos dois operandos na Figura 55. Se um $operando_1$ for carga do transporte e outro $operando_2$ for a sua capacidade, teremos a condição para indicar se o transporte está cheio ou não. Para saber se cabem mais encomendas usamos o operador lógico *menor ou igual* entre a carga atual e a capacidade do transporte. Expressões mais complexas podem utilizar árvores de expressões ligadas através de conectores (*e*, *ou* e *não*).

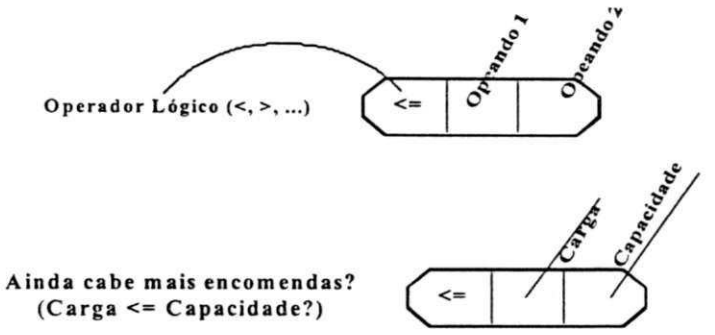

Figura 55: Representação de Condição

Uma restrição define uma condição que sempre deverá ser verdadeira e está associada aos relacionamentos entre os conceitos dos MESOOs. Vamos representar as restrições com um fundo diferente das condições, como pode ser visto na Figura 56. Note neste exemplo que há uma especificação sobre a capacidade de carga de uma moto que nunca deve exceder 10 kg.

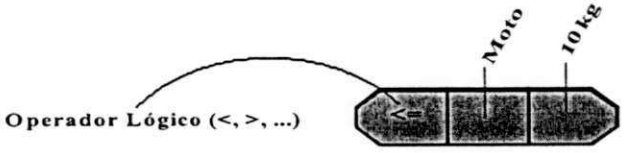

Figura 56: Representação de Restrição

Eventos (Transição de Estados)

Eventos estão relacionados à ocorrência de estímulos que vão iniciar um processo ou atingir uma entidade. A ocorrência de um evento se caracteriza pelo recebimento de uma mensagem em um processo ou entidade e com sua respectiva mudança de estado.

O controle de eventos deve ser feito de forma a monitorar as transições de estados do sistema sendo modelado.

Estado (Valor)

O *estado* representa o valor de uma entidade em um determinado instante na aplicação. A modelagem dos estados é usada para representar os aspectos dinâmicos do sistema sendo especificado. Os estados podem ser representados pelo nome do estado (dado) entre parênteses:

(Carga Cheia) (10 Kg)

(Portão Fechado) (Recepção Operacional)

As especificações nos MESOOs podem conter a indicação de quais valores serão usados como estados: inicial e final.

4.1.1.2 Conceitos Derivados

Os conceitos derivados implementam associações ou combinações dos conceitos base, de forma gráfica através de arcos mostrados na Figura 57, para representar conceitos presentes nos MESOOs. Descrevemos a seguir os seguintes Conceitos Derivados:

- ✓ Parte de
- ✓ Especialização
- ✓ Tem valor
- ✓ Operando
- ✓ Estímulo | Resposta
- ✓ Envia | Recebe

Figura 57: Representação Gráfica dos Conceitos Derivados

As associações entre classes são construídas em função da combinação dos conceitos derivados.

Conceito Derivado Parte de

O conceito derivado *parte de* é utilizado para representar agregações. As agregações existem no domínio do problema e são modeladas através dos MESOOs.

Os conceitos $c_1, c_2 \ldots, c_n$ representam *parte de* outro conceito c_t se a soma das partes compõem o todo ($c_t = c_1 + c_2 + \ldots c_n$). Um exemplo pode ser visto na Figura 58. Observe que os dois processos *calcular custos* e *imprimir previsão de entrega* são partes do todo *receber encomenda*.

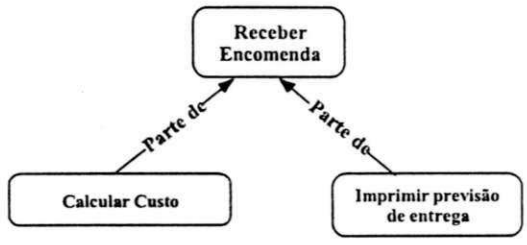

Figura 58: Exemplo do Conceito Derivado *Parte de*

Conceito Derivado Especialização

O conceito derivado *especialização* é utilizado para representar abstrações de outros conceitos. Um conceito c_e é uma especialização de outro conceito mais geral c_g se este acrescentar mais detalhes para uma ou mais entidades deste último. Os conceitos c_g e c_e devem ser do mesmo tipo. O conceito c_e deve acrescentar detalhes específicos a c_g. Um exemplo é uma *Pessoa* que é a generalização de *Cliente*, que vai

entregar a encomenda, e do *Recepcionista* que a recebe (Figura 59). Ou, lendo em outro sentido o diagrama, temos que um *Cliente* e *Recepcionista* são especializações para *Pessoa*

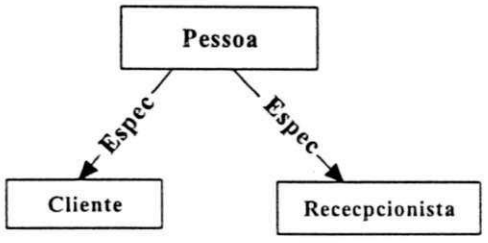

Figura 59: Exemplo do Conceito Derivado *Especialização*

Conceito Derivado Tem valor

O conceito derivado *tem valor* é utilizado para representar os valores que os conceitos básicos (entidade, processo e estado) recebem do domínio do problema. Um conceito *c tem valor v* se *v* for um valor extraído da especificação do problema e deve ser atribuído a *c* como seu valor. Um exemplo é a atribuição da capacidade máxima de transporte da moto (Figura 60).

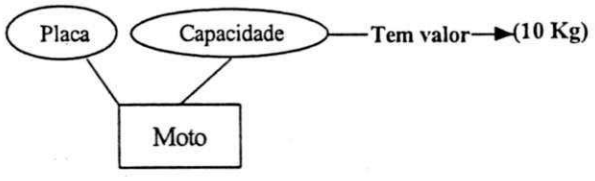

Figura 60: Exemplo do Conceito Derivado *Tem valor*

Conceito Derivado Envia | Recebe

O conceito derivado *envia* disponibiliza uma mensagem, tornando-a ativa, para que um processo ou uma entidade possa receber a informação.

O conceito derivado *recebe* habilita um processo ou uma entidade a receber (processar) uma mensagem. Um exemplo pode ser visto na Figura 61. O cliente envia a solicitação de transporte para a recepcionista.

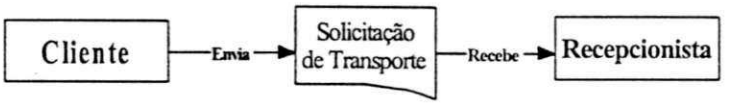

Figura 61: Exemplo do Conceito Derivado *Envia/Recebe*

Conceito Derivado Estímulo/Resposta

O interpretador de pacotes (*IP*) que desenvolvemos funciona como um monitor que vai trabalhar analisando os conceitos básicos e derivados para reproduzir os diagramas dos MESOOs. Na representação gráfica proposta para STL, o *IP* precisa saber como focalizar ou tratar prioridades relacionadas com diagramas que apresentam conceitos relacionados com descrições de características dinâmicas. O *IP* precisa dos conceitos de *Estímulo* e *Resposta* para interpretar uma especificação E_{Am} de uma aplicação A em um MESOO m em seu formato STL Gráfico aqui proposto.

Os conceitos derivados *Estímulo* e *Resposta* capturam os aspectos dinâmicos relacionados com as transições que podem ocorrer entre os conceitos básicos e derivados. O *estímulo* habilita ou ativa um conceito para o qual ele aponta. O *estímulo*↑ é habilitado pelo início da execução do conceito para o qual ele aponta. O *estímulo*↓ é habilitado pelo término da execução do conceito para o qual ele aponta. O significado de *estímulo*↑ e *estímulo*↓ leva em consideração o conceito

para o qual este aponta. Como podemos ver a seguir, se *estímulo*↑ e *estímulo*↓:

- apontar para um conceito do tipo atributo, o significado é "valor torna-se atual (desatualizado)". Isto informa ao *IP* para efetuar a transição fazendo com que o foco agora seja modificado (fazendo com que haja transição no estado do modelo STL Gráfico proposto);
- apontar para um conceito do tipo mensagem, o significado é "a mensagem torna-se ativa (inativa)". Isto informa ao *IP* para efetuar a transição fazendo com que o foco agora seja modificado;
- apontar para um conceito do tipo condição, o significado é "a condição torna-se verdadeira (falsa)". Isto informa ao *IP* para efetuar a transição fazendo com que o foco agora seja modificado;
- apontar para um conceito do tipo processo, o significado é "o processo torna-se ativo (inativo)". Isto informa ao *IP* para efetuar a transição fazendo com que o foco agora seja modificado.

O conceito derivado *resposta* registra o que precisa ocorrer quando a transição em *IP* acontece. A seta *resposta* deve apontar para condições verdadeiras. Se *resposta*↑ ou *resposta*↓ apontar para:

- um conceito do tipo atributo, o significado é "valor torna-se verdadeiro (falso)";

- um conceito do tipo mensagem, o significado é "a mensagem torna-se ativa (inativa)";
- um conceito do tipo processo, o significado é "o processo torna-se ativo (inativo)".

Como podemos ver na Figura 62, é necessária a existência de um monitor (*IP*) que controle as transições que devem ser disparadas. Neste exemplo, a transição será disparada quando a mensagem "adiciona item" for criada, enquanto a recepcionista estiver operacional e a capacidade de carga do transporte não for ultrapassada. O processo "Imprimir Recibo" torna-se ativo fazendo com que o recibo seja impresso.

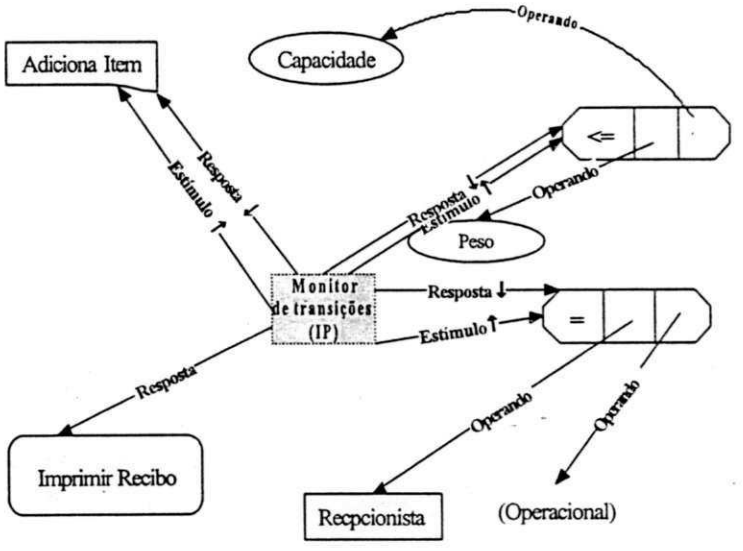

Figura 62: Exemplo de *Estímulo / Resposta*

Conceito Derivado Operando

O conceito derivado *operando* representa o relacionamento entre os conceitos *condição e restrição* e seus operandos.

4.1.2 Transferência entre Ferramentas CASE

Uma especificação E_{Am} de uma Aplicação A, escrita em um MESOO $m \in M$, pode ser descrita através de pacotes de informações p que são compostos de grupos de conceitos base e derivados em STL. Suponhamos, a título de exemplo, que m seja UML e que a especificação E_{Am} a ser transferida seja a agregação de carga e a herança que existe em transporte, como mostra a Figura 63. Um transporte pode ser um carro, ou uma moto, ou um caminhão. Uma carga é composta de um roteiro de entrega e uma ou mais encomendas.

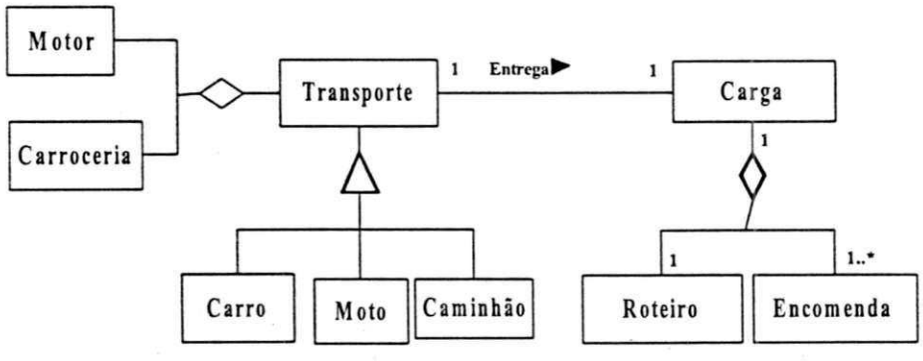

Figura 63: Exemplo de uma Especificação em UML

Observe na Figura 64 que os conceitos de E_{Am} são mapeados para o metamodelo STL através de seus conceitos base e derivados. A cardinalidade é descrita através de uma restrição no relacionamento *parte de*. A associação *Entrega* existente entre

Transporte e *Carga* é mapeada por uma mensagem e associada pelos conceitos *Envia* e *Recebe* do metamodelo STL. A **agregação** utiliza o conceito derivado *parte de*, enquanto a **herança** utiliza o *espec* (especialização) em STL. A cardinalidade é imposta pelo uso de restrições.

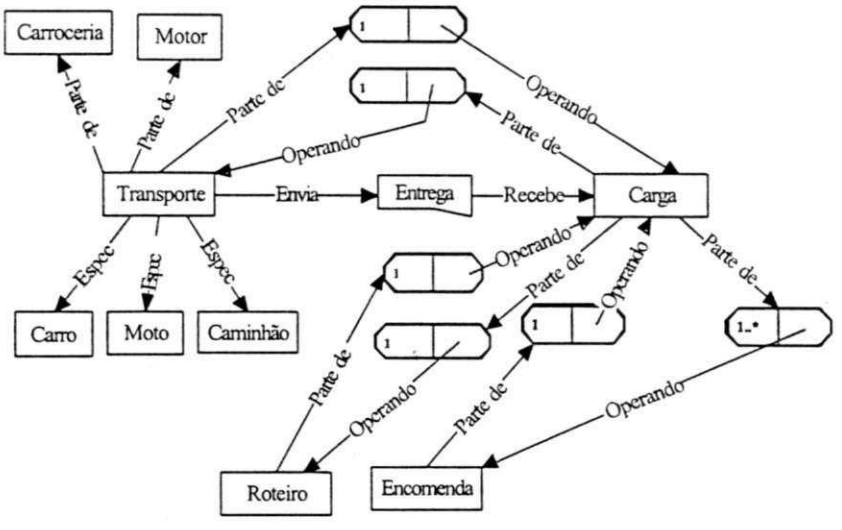

Figura 64: Visualização do Exemplo de Agregação e Herança em STL Gráfica

Como podemos ver na Figura 64, a cardinalidade de *Entrega* "*1 para 1*" está especificada em termos de *Operando* e *Parte de* associados a *Transporte* e *Carga*.

4.1.3 Semântica dos Pacotes em STL Gráfico Proposto

Nesta seção, vamos estabelecer mecanismos para "interpretação" ou "execução" das especificações através dos pacotes STL decompostos em termos de seus conceitos base e derivados. A notação semântica utilizada é similar àquela empregada inicialmente por Maler *et al.* [MALE 91] e depois, também por Davis [DAVI 97].

Os atores principais são as transições de estados dos conceitos básicos e derivados de STL Gráfico proposto que representa uma especificação E_{A_m} de uma aplicação A de um MESOO m. Apenas as especificações que contenham transições poderão ser executadas. Ser "executada" significa que uma especificação começa em um estado inicial específico e muda conforme a chegada de estímulos externos.

A *transição* de estados é central para modelar os aspectos dinâmicos da especificação. No metamodelo proposto deve existir um monitor que controle quais conceitos devem ser disparados, à medida que estes se tornem ativos através de estímulos e respostas. A seguir, definiremos os conceitos base e derivados para formalizar especificações no metamodelo STL proposto.

4.1.4 Formalização do Metamodelo STL Gráfico Proposto

As definições a seguir oferecerão recursos para formalizar o metamodelo STL em termos de uma configuração executável. Trata-se de uma computação ou simulação dos pacotes que são compostos de seus conceitos básicos e derivados e são implementados através do interpretador de pacotes.

Temos as seguintes definições:

Definição 1: Definimos C_m o conjunto dos conceitos presentes em um MESOO $m \in M$.

$$C_m = \{ c \mid c \in m \wedge m \in M \}$$

Onde c é dependente de m podendo apresentar alguns conceitos comuns e outros específicos no universo de M. Exemplo de c

comuns: *objeto*, *classe*, *herança* (generalização / especialização), *agregação*, *associação;* Exemplo de alguns c específicos: *ator*, *exemplo de uso*, *contratos*, *protocolos*, *componentes*, e *nó*.

Definição-2: Definimos $C_{m\text{-}A}$ o conjunto dos conceitos do domínio da aplicação A presentes em um MESOO m. Onde:

$$C_{m\text{-}A} = \{ <c, a> \mid c \in C_m \wedge a \in A \wedge m \in M \}$$

Definição-3: Definimos $D_{m\text{-}A}$ o conjunto dos diagramas d da aplicação A usados para representar graficamente $C_{m\text{-}A}$. Onde:

$$D_{m\text{-}A} = \{ d \mid d = \{ <c, a> \text{ com } c \in m \wedge m \in M \wedge a \in A \} \}$$

Um exemplo de $D_{m\text{-}A}$ pode ser visto na Figura 65 onde temos a representação gráfica de d = {<*classe*, *Encomenda*>, <*associação*, *AlocarTransp*>, <*classe abstrata*, *Transporte*> } em m = *AOO*. O *serviço*, como é chamado em AOO, trata-se de uma *associação* entre as duas *classes* (*Encomenda* e *Transporte*).

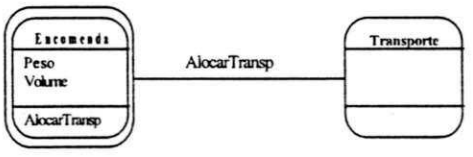

Figura 65: Exemplo de $D_{m\text{-}A}$ (m=*AOO*)

Definição-4: Definimos *converte*$_m$(c, a) uma função que transforma os conceitos $c \in C_{m\text{-}A}$ em conceitos C_{STL} (vide Seções *4.1.1.1-Conceitos Base*, *4.1.1.2-Conceitos Derivados*):

$converte_m: C_{mA} \rightarrow C_{STL} \times A$

$converte_m(c, a) = \{ <c';a> \mid c'$ é o conceito c convertido para STL$\}$

O processo de conversão existe porque os conceitos $c \in C_{mA}$ precisam ser convertidos de sua representação gráfica para o formato textual STL com o objetivo de facilitar a comunicação entre ferramentas CASE.

O exemplo de D_{mA} agora convertido para STL, pode ser visto através da Figura 66. O serviço em *AOO* entre as duas classes (*Encomenda* e *Transporte*) foi traduzido para um *processo* em STL. Um monitor de pacotes *IP* existe para garantir que a informação vai ser reconstruída posteriormente no método específico.

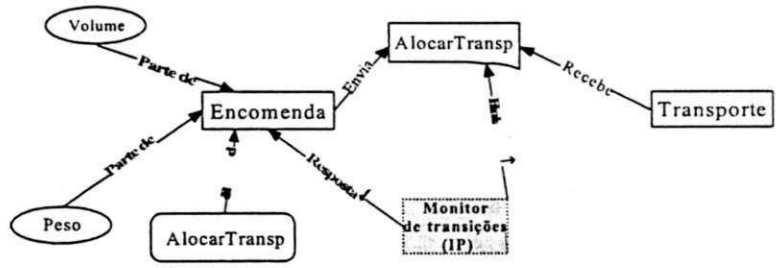

Figura 66: Exemplo de D_{mA} (*em STL*)

Definição-5: Definimos $converte_m^{-1}(c_{STL}, a)$ uma função que transforma os conceitos $c_{STL} \in C_{STL}$ de STL em conceitos $c_m \in C_m$:

$Converte_m^{-1}: C_{STL} \times A \rightarrow C_{mA}$

$Converte_m^{-1}(c, a) = \{ <c', a> \mid c'$ é o conceito STL convertido para $m \}$

Para que ocorra a comunicação entre ferramentas CASE, é necessário um processo de criação (montagem) de pacotes no lado emissor das informações e o processo inverso (desmontagem de pacotes) no lado receptor. Portanto, precisamos monitorar a montagem e desmontagem de pacotes STL.

Definição 6 Definimos o conjunto C_{STL} = {*entidade, processo, mensagem, condição, evento, estado, relacionamento parte de, especialização, tem valor, operando, estímulo, resposta, envia, recebe*} dos conceitos base e derivados presentes no metamodelo STL.

O interpretador de pacotes *IP* operacionaliza o mapeamento entre os conceitos C_{m-1} e C_{STL}. (Vide Seção 4.2 para uma breve discussão sobre o *IP* ou o Anexo A para detalhes).

4.1.5 Exemplo de Mapeamento entre o MESOO AOO e STL Gráfico

Os conceitos $C = \{c_1, c_2, \ldots, c_n\}$ presentes nos diagramas D dos Métodos $M = \{m_1, m_2, m_3, m_4\}$, objetos de estudo deste trabalho, podem ser mapeados diretamente em STL Gráfico proposto, através de seus conceitos básicos e derivados, ou por meio de uma composição destes.

Se tomarmos os diagramas $D = \{d_1, d_2, \ldots, d_4\}$ presentes em m_i (i variando de 1 a 4) podemos observar que: podem existir construções em algum m_i contendo ambigüidade; pode haver diferentes notações d_x e d_y ($x \neq y$) para um mesmo conceito c_x ou

conceitos distintos c_x e c_y sendo representados de uma mesma maneira d_x.

4.1.5.1 Diagramas em AOO (Coad-Yourdon)

Vamos ilustrar um exemplo de mapeamento de AOO [COAD 90] para STL Gráfico proposto.

Como vimos na Seção 2.2, *AOO* é um método de especificação de software orientado a objetos que modela a estrutura e o comportamento dos objetos presentes no domínio do problema. Os diagramas consistem de: objetos que podem conter atributos e serviços; instâncias de conexões entre objetos que mostram como objetos podem cooperar para um usar o serviço de outro através de troca de mensagens. Neste tipo de representação a cardinalidade pode ser explicitada; relacionamentos entre objetos que do tipo *gen-spec* que define hierarquia ou herança entre objetos; e conexões do tipo *todo-parte* que define agregação entre objetos. Podemos pensar em uma forma genérica de representar um diagrama em *AOO* em STL Gráfico, através de seus conceitos básicos e derivados. A Figura 68 representa um diagrama em *AOO* onde dois objetos se comunicam através de *msg* para ativar o serviço *Levantar* o *Portão 1* por onde deve passar o *Caminhão AMN1122*.

Figura 67: Exemplo de Objeto *Caminhão AMN1122* em AOO

A representação do *Caminhão AMN1122* da Figura 67 em STL Gráfico proposto pode ser vista através da Figura 68. Os conceitos básicos *Entidade*, *Processo* e *Atributos* são usados em conjunto com o conceito *Parte de* em STL Gráfico proposto para visualizar o mapeamento entre o MESOO *AOO* e STL. De uma forma genérica, podemos ter objetos ou classes dos diagramas $D \subseteq AOO$ [COAD 90] descritos através de STL Gráfico como ilustra a Figura 69. Um objeto chamado *Nome* possui atributos *{atributo$_i$}* e serviços ou métodos *{métodos$_j$}*.

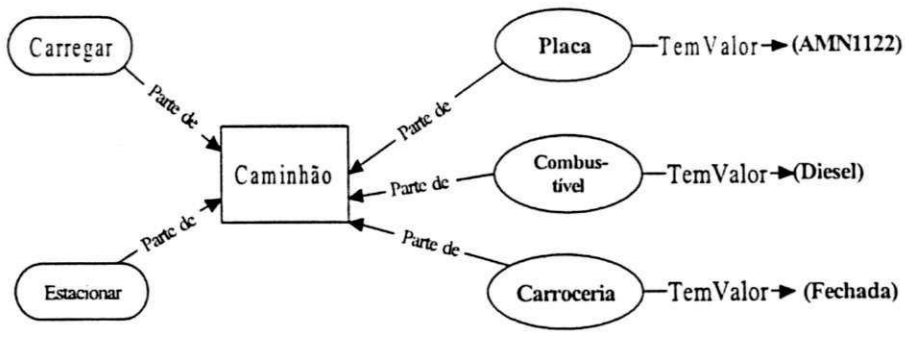

Figura 68: Objeto *Caminhão* de AOO em STL Gráfico Proposto

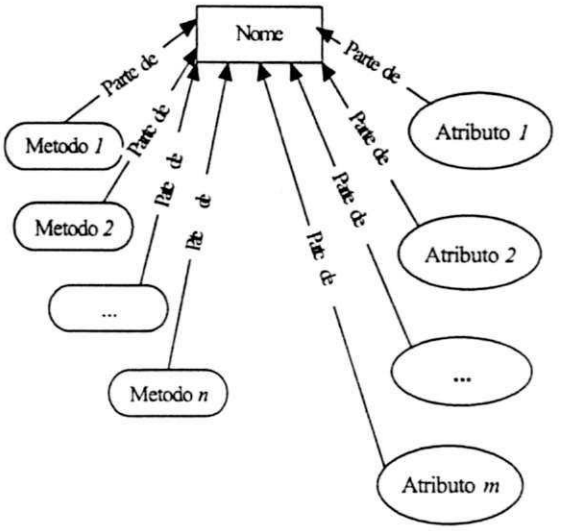

Figura 69: Generalização de Classe em STL Gráfico

Se dois objetos Obj_1 e Obj_2 precisarem se comunicar (vide Figura 12 na página 36) através de uma mensagem *Levantar#1* (levantar o *Portão Via 1*, por exemplo) a representação em STL Gráfica Proposta será como na Figura 70. Neste caso, o interpretador de pacotes STL IP está envolvido para montar a mensagem *Levantar#1* partindo de Obj_1 para Obj_2.

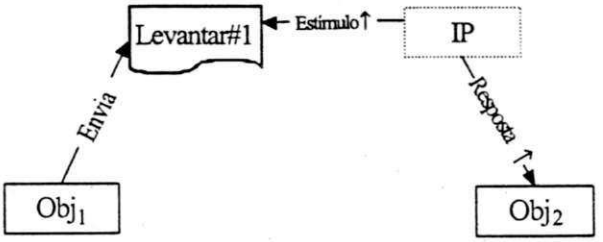

Figura 70: Exemplo de Associação AOO em STL Gráfico Proposto

As estruturas envolvendo Obj_1 e Obj_2 (tais como *Transporte* e *Carro*, vide página 35) em *AOO* do tipo *gen-spec* (herança) são mapeadas para STL Gráfica através do conceito *Espec*, como podemos

ver na Figura 71. Já o relacionamento *todo-parte* em *AOO* é feito em STL através do conceito *Parte de* envolvendo Obj_1 e Obj_2.

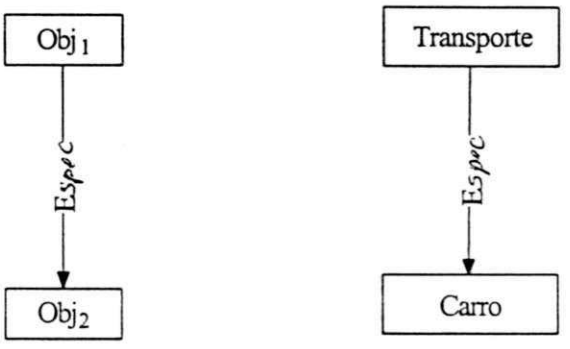

Figura 71: Exemplo de *gen-spec AOO* em STL Gráfico Proposto

Um exemplo de diagrama parcial de *envio de encomendas* pode ser visto na Figura 72. Um Cliente pode enviar uma ou mais encomendas. Estas devem ser alocadas em um transporte de acordo com sua capacidade de carga em função do peso e do volume. A representação correspondente em STL Gráfica Proposta está na Figura 73.

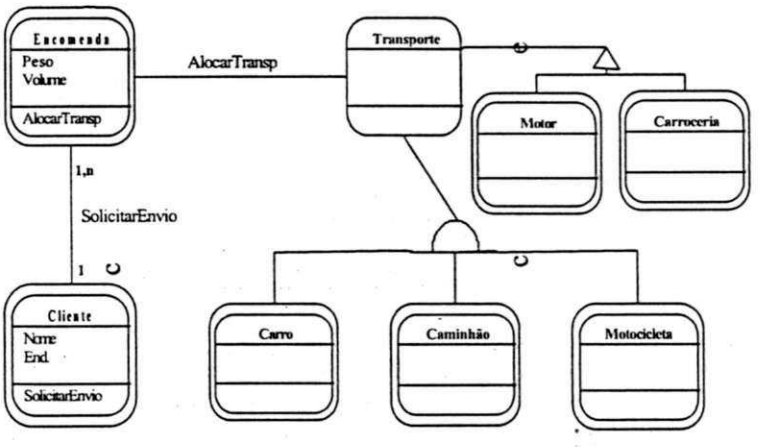

Figura 72: Exemplo Envio de Encomendas em *AOO*

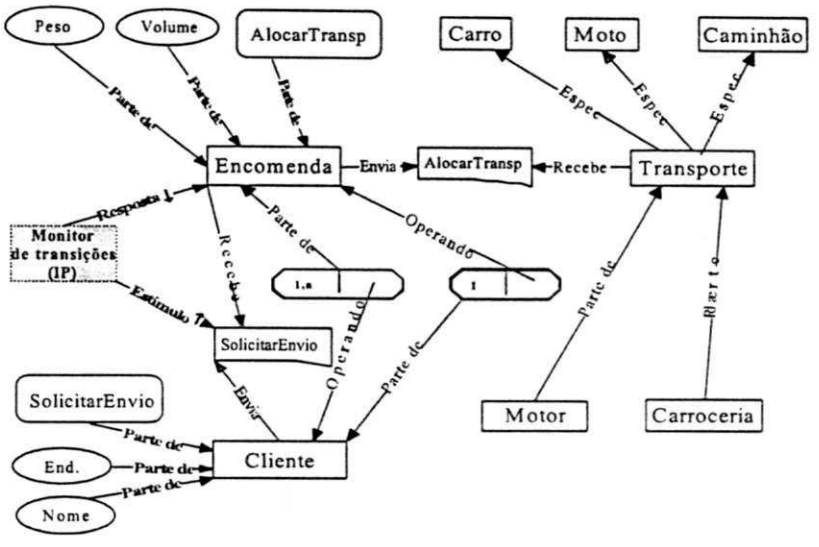

Figura 73: Exemplo Envio de Encomendas em STL Gráfica

Observe que informações adicionais, sobre aspectos dinâmicos da especificação de uma aplicação, podem ser feitas através do monitor de transições *IP* e dos conceitos derivados *Estímulo* e *Resposta* para indicar como montar e desmontar o pacote STL, na fonte e no destino, respectivamente. No exemplo, ilustramos o foco a partir do qual o *Exemplo de Encomendas* será interpretado.

4.2 O Interpretador de Pacotes *IP*

Além de fornecer uma biblioteca de funções que possa ser oferecida às ferramentas CASE para que estas possam incorporar o interpretador STL e que desejarem se comunicar, sem a necessidade de conhecimento prévio da estrutura interna de cada uma das ferramentas, podemos imaginar a computação ou simulação de uma configuração executável *CE* feita através do reconhecimento pelo *lex* e *yacc* das cláusulas STL.

Pode-se identificar os conceitos básicos e derivados do metamodelo de STL na sua forma textual. Os diagramas são transformados em texto, através de seus conceitos básicos e derivados, na ferramenta de origem e são empacotados e enviados para a ferramenta de destino. Lá o pacote é remontado e o diagrama original é recomposto. Assim, é possível visualizar as ações que estão sendo feitas através do reconhecimento dos conceitos de STL. Através da notação STL Gráfica, podemos observar que nas duas especificações (vide Figura 63 e Figura 72), feitas, respectivamente pelos MESOOs m=*UML* e n=*AOO*, vai existir representação comum em STL Gráfica para notações em diferentes MESOOs m e n dos conceitos de herança e agregação utilizados para especificar fragmentos de uma aplicação E_{Am} e E_{An} como ilustra a Figura 74.

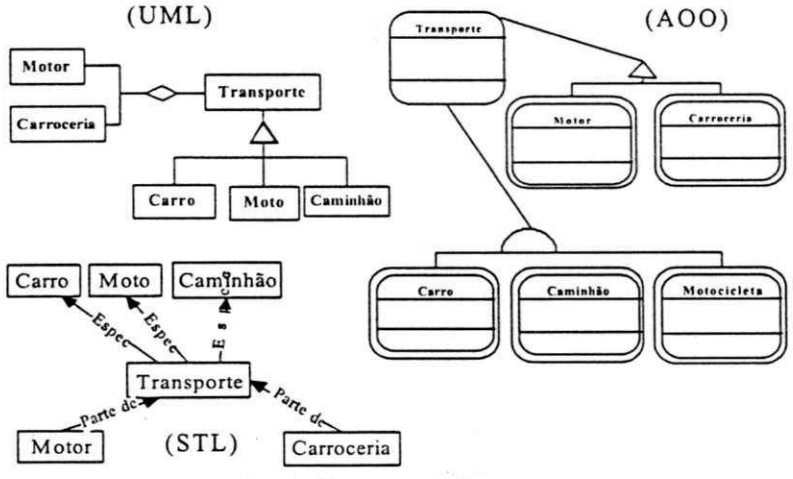

Figura 74: Necessidades de Equivalência em STL Gráfica

A percepção de que existe equivalência entre $E_{Am(UML)}$ e $E_{An(AOO)}$ é feita visualmente, através da STL Gráfica proposta. UML e AOO apresentam notações diferentes para os mesmos conceitos de

herança e agregação. O metamodelo atual de STL não contempla esta visão. Não há como manter informações sobre que conceitos estão sendo mapeados.

O *IP* mapeia $E_{Am(UML)}$ e $E_{Am(400)}$ em uma representação em STL mas não provê nenhum conceito que possa relacionar os conceitos que estão sendo transferidos através de seus pacotes.

No próximo capítulo apresentaremos proposta de um novo metamodelo para STL que possa utilizar o conceito de equivalência no processo de transferência de pacotes de informações entre duas ferramentas CASE.

Capítulo

5

Novo Metamodelo para Transferir Semântica

Este capítulo discute a necessidade de usar um padrão para comparar Métodos de Especificação de Software Orientados a Objeto (MESOOs) e apresenta uma proposta de metamodelo para a Linguagem de Transferência de Semântica (STL).

A abordagem adotada utiliza um novo conceito base STL para comparar diferentes MESOOs.

5.1 Identificando Dificuldades para Transferir Semântica

Os Métodos de Especificação de Software têm evoluído ao longo dos anos. Novas visões ou perspectivas são criadas para melhor especificar o domínio do problema. Com isto, surgem diagramas que materializam novas notações para representar especificações diferentes. Suponha que uma pessoa, durante o processo de

especificação E_{Am} de uma Aplicação A, que esteja usando uma ferramenta CASE FC_{aA} necessite trabalhar em outro sistema operacional que apenas dispõe da ferramenta CASE FC_{bA}. Para reaproveitar E_{Am}, é necessário que as ferramentas se comuniquem através da transferência de semântica. Neste caso, $FC_{a\text{-}Am}$ e $FC_{b\text{-}Am}$ devem suportar m e possuir mecanismos de importação e exportação de informações para que a transferência de E_{Am} seja feita com sucesso para transferir a descrição da especificação da aplicação A. Outras razões podem ser citadas para mostrar a necessidade de comunicação entre $FC_{a\text{-}Am}$ e $FC_{b\text{-}Am}$.

Imagine uma situação onde pode não existir *licença de uso* de $FC_{a\text{-}Am}$ em número suficiente para atender à demanda no primeiro ambiente operacional. Por outro lado, podemos encontrar $FC_{b\text{-}Am}$ em quantidade que atenda à procura, mas que não realiza uma determinada operação desejada (gerar código, por exemplo) ou que a interface não disponha de determinadas facilidades desejadas no segundo ambiente de trabalho. A combinação de $FC_{a\text{-}Am}$ e $FC_{b\text{-}Am}$ pode resultar em um melhor aproveitamento dos recursos disponíveis.

Apesar da implementação de A4O[14] não ser o foco central de nossa tese, fizemos um experimento através da construção de quatro ferramentas CASE para representar cada um dos MESOOs $m \in M$ por equipes de desenvolvedores diferentes. Mais detalhes sobre a implementação pode ser visto no Anexo C. O objetivo foi sentir a dificuldade de se implementar ferramentas CASE com representações internas distintas e como elas poderiam trocar informações.

[14] Artefatos de software desenvolvidos para implementar cada um dos MESOOs com representações internas diferentes.

O resultado demonstrou o uso de diferentes representações internas, independetes do seu formato de armazenamento (que podem utilizar os mais variados SGBDs tais como *Paradox* e *Access*, por exemplo), e que cada equipe focalizou aspectos diferentes quanto à implementação da interface e facilidade de uso na construção dos diagramas. A interface é mais elaborada em um ambiente, em outro há maior possibilidade de representação de mais de um $m \in M$, por exemplo (Veja na Figura 75 e Figura 76 exemplos das interfaces de duas das quatro ferramentas CASE A4O construídas nesta tese). O processo de escolha da ferramenta pode ser sazonal. Em determinadas circunstâncias $FC_{a\text{-}Am}$ pode ser preterida em favor de $FC_{u\text{-}Am}$ ou vice-versa, como já argumentamos anteriormente.

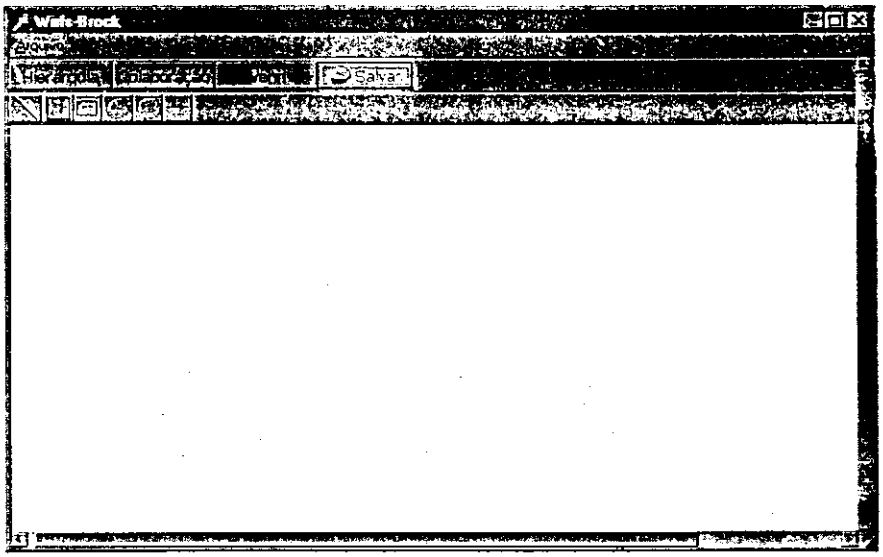

Figura 75: Interface de FC_{aAm}

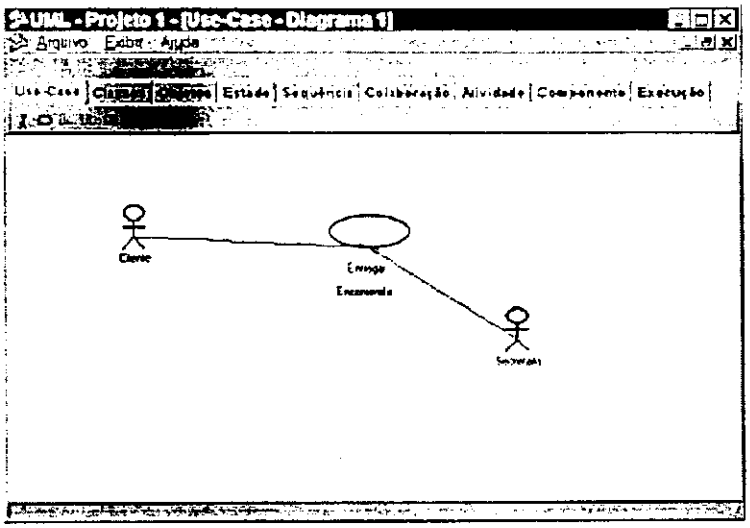

Figura 76: Interface de FC_{Ide}

Se olharmos ferramentas CASE do mercado vamos encontrar mais argumentos para incrementar a dificuldade do processo de escolha de uma ferramenta para automatizar $\mathcal{M}$ na especificação de uma Aplicação $\mathcal{A}\,E_{Am}$ Podemos citar dois exemplos na área comercial: Rational Rose (Figura 77) e Together (Figura 78).

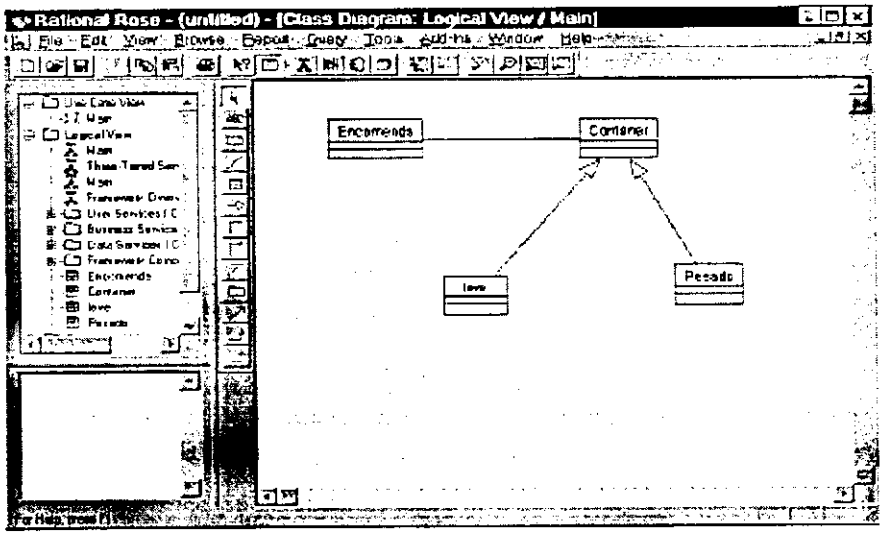

Figura 77: Interface de Ferramenta CASE Comercial 1

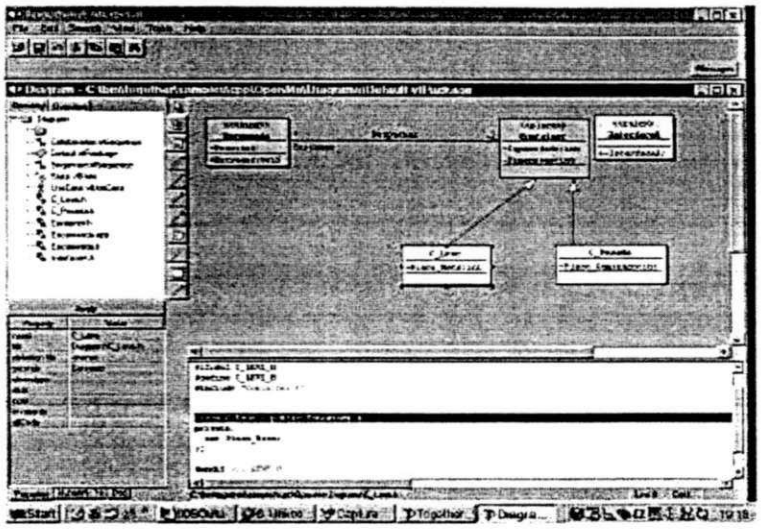

Figura 78: Interface de Ferramenta CASE Comercial 2

Independente do valor (quanto se paga em relação à usabilidade em termos de funcionalidade e à política de licenças de uso), existem outros aspectos igualmente importantes na hora da escolha de uma ferramenta, tais como quais recursos computacionais de hardware e software (capacidade de armazenamento em disco, velocidade da unidade central de processamento e quantidade de memória mínima exigida). A interface tem um forte apelo principalmente se aliar funcionalidade com desempenho. Um exemplo pode ser visto na Ferramenta CASE Together (Figura 78). Ela gasta menos tempo com a carga e pode gerar código para uma linguagem de programação destino à medida que os diagramas são gerados. Por outro lado, como podemos ver na tela de abertura (Figura 77) do Rational Rose, há opções para personalizar as especificações para um determinado domínio de aplicações como: acesso a uma base de dados isolada; a construção de aplicações transacionais; a construção de aplicações cliente-servidor para a Internet. Com certeza estes

exemplos e argumentos não esgotam a necessidade de cada vez mais as ferramentas CASE trocarem informações semânticas para que o melhor de cada ambiente seja explorado pelos usuários.

Estamos falando em transferência de semântica entre ferramentas CASE diferentes que utilizam um mesmo MESOO $m \in M$. Se tomarmos também $n \in M$, $m \neq n$, teremos que exigir que as ferramentas $FC_{a,A}$ e $FC_{b,A}$ suportem também n, além de m, e que permitam transferência de semântica entre os MESOOs m e n.

De uma forma geral, uma especificação $e \subset E_{Am} = \{e \mid e \subset m\}$ depende da definição dos conceitos $C_m = \{c_1, c_2, c_3, ..., c_n\}$ utilizados para construir E_{Am}. Em um mesmo MESOO $m \in M$, podemos encontrar conceitos utilizados de forma ambígua. Ou seja, para $i \sqsubset j$ temos c_i e c_j descrevendo uma mesma especificação $e \in E_{Am}$. Para dificultar ainda mais o processo de transferência de semântica, podemos imaginar a necessidade de visualizar E_{Am} em uma ferramenta CASE que suporte apenas o MESOO $n \in M$. Neste caso, podemos encontrar dificuldade de mapeamento envolvendo algum $c_i \circ C_m \mid \neg\exists c_j \circ C_n$ que permita a visualização da especificação E_{Am} em uma ferramenta CASE que apenas suporte E_{An}.

O metamodelo de STL permite o mapeamento dos Conceitos C_m presentes em uma especificação de uma aplicação E_{Am} em termos de Pacotes STL $p \circ P_m$. A forma textual de descrição de pacotes permite que um interpretador seja anexado às Ferramentas CASE para fazer o mapeamento representado por $Int_{FC}<E_{Amo}\ P_m>$. Definimos $Int_{FC}<E_{Amo}\ P_m>$ como sendo a versão automatizada da gramática STL do seu metamodelo, construída em função dos

comandos *lex* e *yacc*, que interpreta os pacotes P_m que correspondem à especificação da Aplicação E_{Am}. Esta operação transforma $C_m \subseteq E_{Am}$ em $p \circ P_m$. Suponhamos que o MESOO m seja UML e que C_m = *{agregação, generalização, associação}*.

Para ilustrar o lado operacional do mapeamento, seja o *Diagrama de Classes em UML* que aparece na Figura 79 para representar uma especificação E_{AUML} para descrever como fazer entrega de encomendas através de um motociclista.

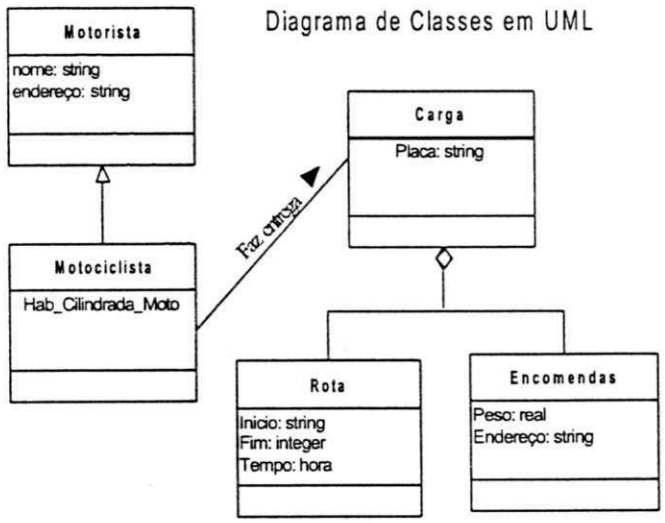

Figura 79: Exemplo de Diagrama de Classes em UML

Construímos um software que possibilita a verificação do mapeamento dos conceitos UML para pacotes STL na sua forma textual, vide Figura 80. A informação em E_{AUML} é textualizada em função dos conceitos do metamodelo de STL, através de suas cláusulas.

/* Comment Catalog of S_Packet contents:

1 diagram Collection	Ex_Figura 79;
5 Object DataTypes	Motorista,
	Motociclista,
	Carga,
	Rota,
	Encomendas;
9 entity attribute DataParts	m_nome, m_end, mo_hab, c_placa, r_inicio,
	r_fim, r_tempo, e_peso, e_endereço;

*/

S_Packet		**E_UML_Exemplo**
	has subject	"Exemplo de E_{AUML} e Pacotes $_{STL}$";
	has stl version	"1.0";
	has content version	"Versão 1.0";
	has creator	"Álvaro F. C. Medeiros";
	has description	" E_{AUML} exemplo para mostrar Pacotes $_{STL}$!";
Collection		**Figura1**
	has label	"Diagrama de Classes em UML";
	has description	"Exemplo de E_{AUML}";
	has standard usage	erd;
	groups objects	Motorista, Motociclista, Carga, Rota,
		Encomendas;
Object		**Motorista**
	has label	"Motorista";
	has subtype object	Motociclista;
	encapsulates action	m_revisar_veiculo;
	has component datapart	m_nome,
		m_endereco;
	offers dataitem	m_nome,
		m_endereco;
	is pictured with graphicsymbol	"uml_class.bmp";

Object **Motociclista**

```
                has label                       "Motociclista";
                is subtype of object            Motorista;
                encapsulates action             mo_abastecer_veiculo;
                has component datapart          mo_hab_cilindradas_moto;
                offers dataitem                 mo_hab_cilindradas_moto;

                is pictured with graphicsymbol  "uml_class.bmp";
ConnectionPath                  motorista_Motociclista
                Is connectiontype               data;
                Is encapsulated in object       motorista;
                is grouped into collection      Figura1;
                is pictured with graphicsymbol  "uml_heranca.bmp";
( ... )
pe_mark     /* marca de fim de pacote */
```

Figura 80: Exemplo de $E_{Am\ (UML)}$ em STL

A dificuldade de transferência de semântica foi aqui verificada através da execução da interpretação de E_{Am} e E_{An} por meio de um interpretador de pacotes *IP* construído em função do *lex* e do *yacc*. Ao ativar o IP para interpretar E_{Am}, para o MESOO $m=UML$, temos a textualização ou a construção de um pacote em STL, parecido com o que foi apresentado na Figura 80, que será enviado para outra ferramenta CASE. Temos outra representação em STL para E_{An}, onde $n=UML$, sem, no entanto, termos equivalência entre conceitos. A ativação do interpretador de pacotes STL trata E_{Am} e E_{An} isoladamente. Não há como fazer mapeamento entre eles pois os conceitos *base* e *derivados* de STL não contemplam nenhum mecanismo que registre equivalência ou comparação entre os conceitos.

O metamodelo de STL limita o uso dos pacotes p, por não permitir o uso mais abrangente em termos comparativos, em relação ao processo de mapeamento dos conceitos c_i e c_j presentes em

especificações E_{Am}. De uma forma geral, dado duas especificações E_{Am} e E_{An}, STL não viabiliza $FC_{aA}<C_m, P_{ma}>$ e $FC_{bA}<C_n, P_{nb}>$ que permita uma análise sobre quais conceitos a e g se relacionam ou que permitam identificar sobreposições de semântica de conceitos.

A notação $FC_{aA}<C_m, P_{mx}>$ representa uma Ferramenta CASE de um fabricante "a" que esteja usando o software construído em função do *lex* e do *yacc* e posto em uma biblioteca para ser *linkeditado* para fazer parte do ambiente operacional como se fosse uma "caixa preta". Já $<C_m, P_{mb}>$ informa como mapear os conceitos C_m para pacotes STL na ferramenta linkeditada com a biblioteca produzida pelo *lex* e *yacc* produzida pelo distribuidor "b".

5.2 Compatibilidade entre Visões

Hoje, existem muitos MESOOs. Booch e Rumbaugh [BOOC 95a] estimam que o número em 1995 era superior a 70. Os métodos são compostos de idéias comuns como generalização, especialização, associação, cenários, etc., e divergem com respeito a relacionamentos (como por exemplo, *responsabilidade* e "*roles*"). A proliferação de métodos sugere que as diferenças são superficiais e que geralmente se localizam na representação gráfica, na notação ou terminologia, mas com pouca divergência no contexto semântico dos conceitos [TEXE 98].

Há ainda o impacto da escolha de que conceitos c_1 e c_2 usar em um mesmo MESOO $m \in M$, para construir especificações de uma aplicação E_{Am} para representar a visão de interesse no domínio do problema ou no cenário. A especificação E_{Am} pode ser decomposta

com base em c_1, c_2 com o objetivo de minimizar ou maximizar a sobreposição dos conceitos, numa visão comparativa entre o uso de c_1 e c_2 para representar E_{Am}. Um exemplo clássico seria a escolha entre associação e agregação.

Apesar de cada MESOO m estabelecer uma metodologia de uso próprio visando manter consistência no uso de seus conceitos, é possível escrever E_{Am} cuja sobreposição de c_1 e c_2 em STL pode ser minimizada (área 1 na Figura 81) ou maximizada (área 2).

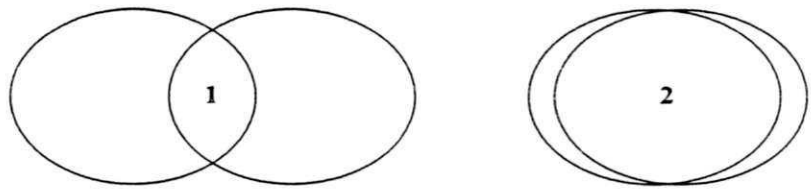

Figura 81: Dependência da Representação do Modelo

Suponhamos que para modelar as ações do cenário sejam utilizados dois conceitos de um MESOO tais como Redes de Petri (c_1) e Diagramas de Fluxo de Dados (c_2), por exemplo. Vamos notar que haverá uma sobreposição ou visão de equivalência em STL menor (área 1). Neste caso, em STL os conceitos base serão diferentes. Por outro lado, se compararmos herança simples com múltipla, como ilustra a Figura 82, teremos uma visão de equivalência com maior sobreposição (área 2). Há uma divergência menor em como os conceitos que representam as heranças simples e múltiplas. Neste caso, detalhes envolvendo conceitos derivados em STL representam as mudanças necessárias para indicar as diferentes cardinalidades de c_1 e c_2. As visões ou comparações têm sido estudadas na bibliografia. [FOWEL 95, IIVA 95, KUCE 94, LUBA 93]

Uma análise parcial das representações de um mesmo problema em mais de uma metodologia "*A survey of influential object-oriented analysis methods*" feita por Mitchell Lubar [LUBA 93] evidencia o impacto da escolha do modelo na representação de nosso metamodelo STL. Se tomarmos um mesmo problema e fizermos sua decomposição baseada em duas abordagens, orientada a ações e orientada a objetos, o resultado de nossa representação STL será muito diferente no caso de modelos de representação diferentes, como era de se esperar.

O Padrão IEEE 1175 oferece mecanismo para comunicação entre ferramentas CASE que utilizem um mesmo MESOOs $m \in M$ através do empacotamento do conjunto de Conceitos C (generalizações, especializações, agregações e outras associações) para os conceitos base e derivados de STL. Aproveitando os pacotes STL podemos estender a comparação de conceitos c_1 e c_2 presentes em diferentes MESOOs $m_1 \in M$ e $m_2 \in M$. Suponhamos que m_1 possa permitir associações que m_2 não aceite. Por exemplo, a Figura 82 mostra uma possível diferença de m_1 em relação a m_2. Neste caso, podemos dizer que um MESOO suporta herança múltipla e o outro não.

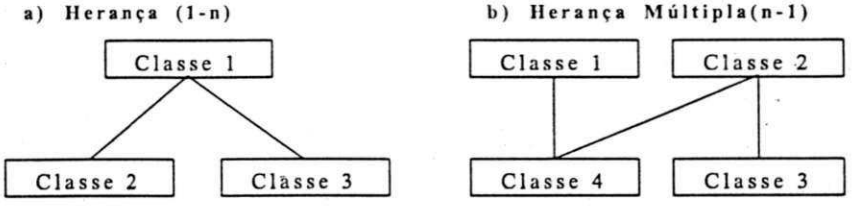

Figura 82: Cardinalidade nas Associações

A região *(1)* da Figura 81 seria um indicador de compatibilidade entre os MESOOs m_1 e m_2. Uma organização poderia adotar um critério com relação a adoção de novos métodos baseado na compatibilidade entre os modelos. Este percentual seria calculado em função do número de conceitos c_1, c_2, ..., c_n dos MESOOs m_1 e m_2 e de como eles podem ser mapeados no novo metamodelo STL em termos da visão do conceito de equivalência.

Analisar MESOOs m_1 e m_2 balizados em uma representação comum em STL leva-nos a comentar os aspectos de portabilidade dos MESOOs.

5.3 Portabilidade dos Métodos

Podemos tomar emprestada a idéia de portabilidade da Linguagem C [KERN 78] e do Sistema Operacional Unix [MEDE 94] ou do Ambiente Java. O compilador C consegue gerar código a partir de um mesmo programa fonte para executar em diferentes plataformas de hardware, após a compilação destes nas plataformas destino. No caso de Java, a complexidade fica por conta de interpretadores ou *máquinas virtuais Java* que devem existir para interpretar um código intermediário entre fonte e linguagem de máquina.

O fato é que a identificação e a separação das partes afetadas na transportabilidade faz com que o Unix seja "facilmente" transportado de uma máquina para outra. O que estes programas têm em comum é a separação das funcionalidades básicas das dependentes de hardware. O compilador C não precisa se preocupar com as

diferenças entre as plataformas, pois existe uma biblioteca padrão com sintaxe e semântica definidas. No caso do sistema operacional, existe o núcleo que congrega o software dependente de hardware, e o esforço para transportar o sistema operacional concentra-se nesta parte. O código fonte é um só. As diferenças são acomodadas através de construções condicionais presentes na própria linguagem.

Outras abordagens baseadas em portabilidade, mais recentes, surgem como tendência para minimizar a complexidade de gerenciamento de diferentes configurações de hardware nas organizações. Podemos citar a proposta da Linguagem Java como exemplo para garantir a portabilidade dos seus programas nas várias plataformas hoje disponíveis no mercado. Observe que, neste caso, o código fonte é único. Diferente da Linguagem C, que utiliza compilação condicional, os programas Java são compostos de *bytecodes* (representação canônica) e as diferenças existentes em cada ambiente operacional, onde o programa será executado, são acomodadas através das *máquinas virtuais* (os interpretadores/compiladores) locais a cada configuração específica. Observe que em ambos os casos os conceitos básicos e os conceitos dependentes do hardware foram tratados para se conseguir portabilidade.

Podemos aplicar esta idéia de selecionar conceitos básicos e derivados de nosso metamodelo e aplicar aos MESOOs. Com esta separação, precisaremos de uma forma de medir ou comparar especificações escritas em métodos distintos.

A aplicabilidade ou escolha de um MESOO está vinculada à sua capacidade de representar cenários ou domínios do problema de

interesse. Conhecer os conceitos nos MESOOs não é o suficiente. Saber quais são os seus pontos fortes e fracos é fundamental. Encontrar o MESOO "certo" demanda tempo para o aprendizado dos conceitos e sua visão comparativa em relação aos demais disponíveis.

5.4 Apresentação do Novo Metamodelo

O metamodelo proposto é composto de um **novo conceito derivado** chamado **equivalência** que poderá ser usado pelo interpretador de pacotes *IP* para fornecer informações sobre equivalência de conceitos C_m e C_n de um mesmo MESOO m ou entre dois diferentes m e n. O objetivo é relacionar conceitos durante o processo de mapeamento entre MESOOs, extendendo STL para favorecer um ambiente de comunicação mais favorável ao compartilhamento de especificações entre ferramentas CASEs produzidas por fornecedores distintos.

O mecanismo normal de construção de pacotes textuais Px, onde x indica que conceito C_m ou C_n está sendo empacotado ou emviado, em STL continuará a ser feito pelo *IP*. O que existirá será um catálogo de equivalência $Eq_{i,j}$ que será construído para informar ao *IP* sobre a equivalência entre os conceitos i e j nas especificações originais dos MESOOs m e n que contêm esses conceitos.

Suponhamos que o par $<E_{Ami}, P_x>$ represente o mapeamento entre o conceito i do MESOOs m para o pacote STL P_x, e que o par $<E_{Anj}, P_y>$ represente o mapeamento entre o conceito j do MESOOs n para o pacote STL P_y. Sabendo-se que existe equivalência

Eq_{ij} entre os conceitos i e j nas especificações originais dos MESOOs m e n temos que os pacotes produzidos por $<E_{Am}, P_x>$ e $<E_{An}, P_y>$ serão equivalentes (total ou parcialmente) (P_x *eq* P_y).

5.4.1 Introdução ao Conceito de Equivalência

No metamodelo da Seção Representação Gráfica do Metamodelo STL estamos propondo a inclusão do conceito que chamamos de **Equivalência**.

Estamos interessados em definir equivalência entre conceitos e utilizar o mecanismo de transferência de semântica para podermos representar conceitos equivalentes no domínio das especificações de uma aplicação E_{Am} escrita em um dos MESOOs m.

Usando o metamodelo da Seção 4.1.1 (O Metamodelo de STL) podemos criar uma representação gráfica de ←→ para indicar que um conceito equivale a outro. Assim, podemos dizer que parte de uma especificação e_1 equivale a outra e_2. Como podemos ver na Figura 83, e_1 = ao estado *(carga completa)* e e_2 = condição *capacidade >= 10 kg*. Introduzimos, através das definições a seguir, o conceito de equivalência em uma notação mais formal.

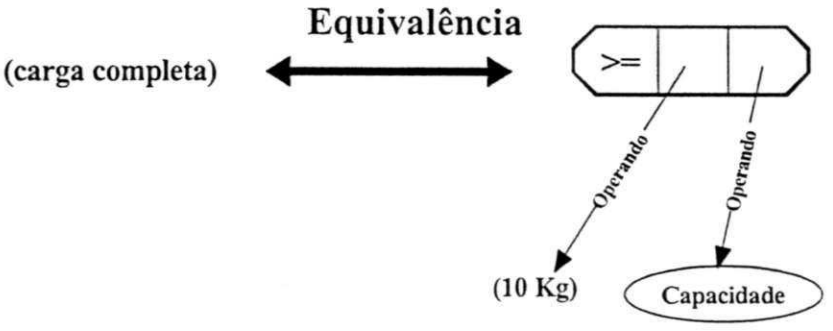

Figura 83: Exemplo do Conceito de Equivalência

As definições de *empacota* e *monta* foram utilizadas para construção do interpretador de pacotes *IP* e utilizam *converte* vista na Seção 4.1.4).

Definição-7: Definimos $empacota_m(c, a)$ uma função que transforma os conceitos $c \in C_{mA}$ de uma aplicação $a \in A$ em pacotes STL $p \in P$. Estes últimos utilizam os conceitos STL (vide Seções *4.1.1.1-Conceitos Base*, *4.1.1.2-Conceitos Derivados*) para construir p em termos de C_{STL}:

$$empacota_m: D_{mA} \rightarrow C_{STL} \times A$$

$$\langle c, a \rangle \rightarrow \langle converte_m(c, a), a \rangle$$

Definição-8: Definimos $monta_m(p, a)$ uma função que transforma os conceitos descritos em pacotes STL $p \in P$, que converte conceitos C_{STL} em C_{mA}, permitindo o desempacotamento e reconstrução de D_{Am}:

$$monta_m: C_{STL} \times A \rightarrow D_{mA}$$

$$\langle c, a \rangle \rightarrow \langle converte_m^{-1}(c, a), a \rangle$$

Definição-9: Definimos $escopo_m(c, a)$ o conjunto de conceitos em C_{STL} que represente um conceito em C_{mA}:

$$escopo_m: C_{STL} \times A \rightarrow D_{mA}$$

$$escopo_m<c_{stl}, a> \rightarrow \{ s \mid s = <c', a> \wedge c = converte(c', a) \}$$

Podemos visualizar $escopo_m(c_{stl}, a)$ em STL Gráfica Proposta através de um retângulo com linhas pontilhadas para representar o mapeamento de c_{stl} e a em D_{mA}. A Figura 84 ilustra como o escopo pode ser delimitado para focalizar o conceito $c \in C_{mA}$ de interesse que corresponde a um a em m igual a herança no caso (a) ou agregação em (b).

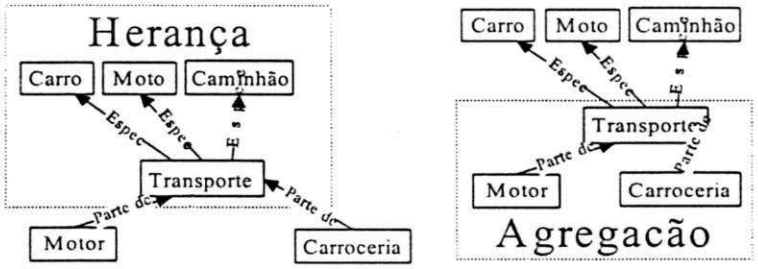

Figura 84: Representação de *Escopo* em STL Gráfica

Definição-10: Definimos o *catálogo de escopos* todos os pares $<e, m>$, onde $e \in Esc_A$ e $m \in M$, que representam os escopos da aplicação A em D_{mA}. Portanto:

$$Esc_{mA} = \{ escopo_m<c_{stl}, a> \mid c_{stl} \in C_{STL} \wedge a \in A \}$$

Definição-11: Definimos que dois escopos são equivalentes em MESOOs distintos se, e somente se, estes representarem um mesmo escopo em STL. Temos que:

$$\approx Eq: C_{mA} \times C_{nA} \rightarrow \{ verdadeiro, falso \}$$

É uma relação entre escopos $e_1 \in C_{mA}$ e $e_2 \in C_{nA}$ tal que $e_2 \approx e_2 \Leftrightarrow \forall s_1 \in e_1 (\exists s_2 \in e_2 \wedge \exists s_0 \in C_{STL} \mid s_0 = empacota_m(c, a) \wedge s_1 = monta_m(s_0, a)) \wedge \forall s_2 \in e_2 (\exists s_1 \in e_1 \wedge \exists s_0 \in C_{STL} \mid s_0 = empacota_m(c, a) \wedge s_2 = monta_m(s_0, a))$.

5.4.2 Aplicação do Conceito de Equivalência

O Catálogo de escopos é uma Biblioteca de Equivalência (*BE*) que deve ser construída para cada $m \in M$ no momento da construção dos pacotes STL. Cada conceito de m deve ser descrito apenas a primeira vez que aparecer na especificcação E_{Am} de uma aplicação A.

A ferramenta *FC* deverá incorporar os artefatos de software construídos em função do *lex* e do *yacc* possibilitanto o acesso ao monitor de equivalência de pacotes. Assim, será possível visualizar como E_{Am} e E_{An} se comportam em relação à equivalência de conceitos ou sobreposição no mapeamento de conceitos de m e n para STL, levando em consideração o *GE* (a quantidade de pacotes transferidos com equivalência, divididos pelo total de pacotes transmitidos em cada MESOO). Desta fórma, podemos utilizar informações sobre a base de especificações legada para tomar decisões de que MESOO melhor se aplica a um domínio de problema.

Outra aplicação possível será fazer com que *FC* possa recuperar pacotes STL que descrevem E_{Am} e poder representa-los usando a notação E_{Am} desde que os conceitos *i* e *j* satisfaçam $Eq(i, j)$, e ambos estejam catalogados na *BE*.

A convergência vai depender da natureza dos conceitos existentes nas visões ou modelos dos MESOOs e da escolha destes para construir as especificações *EAm* e *EAn* da aplicação *A*. Um Diagrama de Caso de Uso em UML, como ilustra a Figura 85, mostra a visão funcional de um cenário do despacho de cargas.

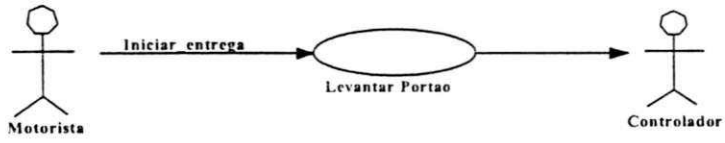

Figura 85: Diagrama Simplificado de Despacho da Carga em UML

Os conceitos de atores e exemplos de uso, ao serem empacotados para STL, podem ser vistos na representação gráfica STL na Figura 86.

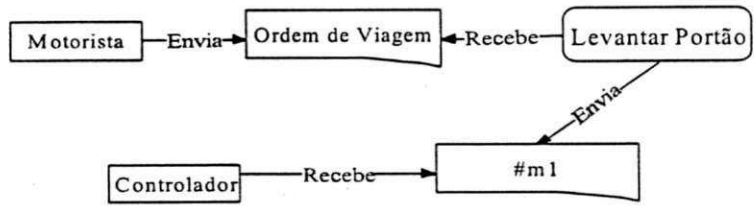

Figura 86: Diagrama Simplificado de Despacho da Carga em STL

Se tomarmos uma outra visão de um mesmo MESOO (UML), utilizando o Diagrama de Seqüência para especificar qual a seqüência de objetos deveria implementar a funcionalidade da Figura 85, teremos a Figura 87.

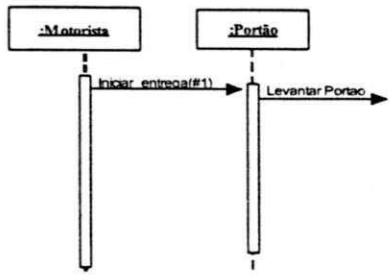

Figura 87: Diagrama Seqüência de Despacho de Carga em UML

Os conceitos de objetos, que tem seu tempo de vida representado em STL por *#E1* e *#E2*, quem enviam mensagens para outros objetos em uma seqüência para completar uma operação são representados graficamente em STL através da Figura 88.

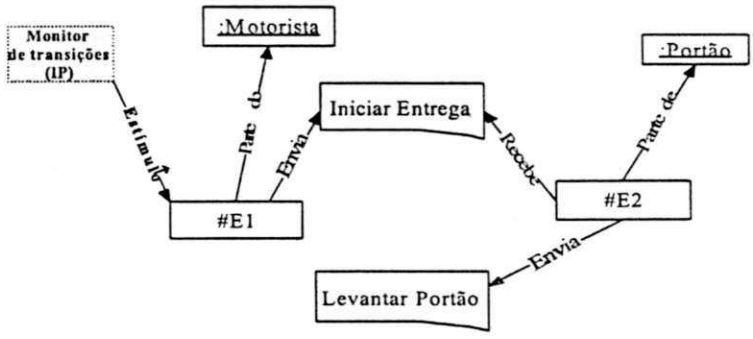

Figura 88: Diagrama Seqüência de Despacho de Carga em UML STL

Com estes exemplos, queremos enfatizar que é necessário haver equivalência entre conceitos no catálgo de escopos para que o GC seja maximizado para um MESOO específico. Nos diagramas acima não houve equivalência por se tratar de visões diferentes. O mesmo aplica-se a MESOOs que apresentem diferentes visões.

5.5 Extensão ao Metamodelo STL

Estamos propondo uma nova abordagem para o emprego do padrão IEEE 1175. Queremos que o mecanismo de troca de pacotes entre ferramentas CASE seja também utilizado para fins de comparação entre conceitos dos métodos de especificação de software orientados a objetos.

Sejam E_{Am} e E_{An} duas especificações de uma aplicação A escritas através dos MESOOs m e n, e p_i e p_j os pacotes que correspondem, respectivamente, ao mapeamento de E_{Am} e E_{An} para o metamodelo STL proposto. Estamos propondo que $Eq(i, j)$ seja usada como parte das cláusulas STL.

A operacionalização de $Eq(i, j)$ se dará por meio da atualização do Interpretador de Pacotes (*IP*) incluindo uma nova cláusula, como ilustra a Tabela 6 e a descrição da implementação do *IP* no Anexo A.

Tabela 6: Nova Cláusula *Equivalence*

Nome_conceito:	**Equivalence.**
Significado:	Agrupa um subconjunto de instâncias de conceitos equivalentes em um pacote STL T.
Apresentação_texto_conceito:	
Palavra-chave:	Equivalence
Identificador_sentença:	EquivalenceID
Possíveis_atributos:	
Has label	Cadeia de Caracteres "rótulo"

Has description	Cadeia de Caracteres "descrição"
Has external usage	Explanação
Possíveis_relacionamentos:	
Has component collection	CollectionID {, CollectionID}
Is component of collection	CollectionID {, CollectionID}
Groups actions	ActionID {, ActionID}
Groups condiction	CondictionID {, CondictionID}
Groups connectionpath	ConnectionpathID{, ConnectinpathID}
Groups datakey Groups datapart	Datakey {, DatakeyID} DatapartID {, DatapartID}
Groups dataitem	DataitemID {, DataitemID}
Groups datarole	DataroleID {, DataroleID}
Groups datastore	DatastoreID {, DatastoreID}
Groups datatype	DatatypeID {, DatatypeID}
Groups dataview	DataviewID {, DataviewID}
Groups eventitem	EventitemID {, EventitemID}
Groups eventtype	EventtypeID {, EventitemID}
Groups state	StateID {, StateID}
Groups statetransition	StatetransitionID {, statetransitionID}
Is pictured with graphicalsymbol	GraphicsymbolID {, GraphicsymbolID }

O novo conceito implicará uma modificação nas gramáticas fornecidas ao *lex* e ao *yacc*. Um novo token deverá ser inserido para representar o conceito de equivalência. Uma nova sentença deverá ser acrescida ao arquivo *stl.y* para que o *yacc* reconheça o novo conceito proposto.

Apresentamos a seguir a visão parcial do texto da modificação em nosso *inter_stl*.

```
...
%token THREAD_EQ_TOKEN
...
%%%
...

Equivalence : THREAD_EQ_TOKEN IDENTIFIER
thread_eq_attributes thread_relations
                    { resolve_relations($2);
thread_eq+=store_symbol($2,"ThreadEq"); printf("ThreadEq ->
%s\n",$2); }
                    ;

thread_attribute : has_label
                 | has_description
                 | has_standard_usage
```

```
                    | has_external_usage
                    ;

thread_relations    : thread_relation ';'
                    | thread_relation ';' thread_relation
                    | collection_relation
                    ;

thread_relation     : groups_datatype
                    ;
```

Mais detalhes sobre a arquitetura do comando *inter_stl*, construído por nós nesta tese para validar pacotes STL nas ferramentas CASE de diferentes distribuidores, encontram-se no Anexo A (Interpretador de Pacotes STL).

5.5.1 Utilidade do Metamodelo ao Longo do Processo de Desenvolvimento

Com o Metamodelo proposto, o Padrão IEEE 1175 passa a contar com um conceito que pode descrever equivalência entre visões ou descrições STL E_{Am} que refletem as Especificações de Aplicações escritas em um MESOO $m \in M$.

Ferramentas CASE utilizadas nas fases de reconhecimento das necessidades do usuário e no gerenciamento do processo produtivo poderão trocar informações com outros utilitários mais

técnicos ligados à implementação específica de um MESOO m, voltados para as últimas fases do processo de desenvolvimento de software, que possuam MESOOs que suportem diagramas ou representação gráfica das funcionalidades especificadas em E_{Am}. Assim, será possível integrar as informações de E_{Am} com o objetivo de gerenciar o processo de desenvolvimento. Dados coletados em uma ferramenta poderão ser utilizados em outra. Basta que ambas utilizem o interpretador de pacotes STL disponível em uma biblioteca de funções para trocar informações.

Finalmente, a abordagem proposta na construção do metamodelo pode ser usada para analisar as descrições STL em termos de quais cláusulas (vide Anexo B) foram utilizadas para representar E_{Am}. Um especialista em STL poderá utilizar este conceito para decidir que MESOO escolher para ser utilizado no desenvolvimento de um novo projeto [MEDE 96, 98].

No próximo capítulo apresentaremos nossa conclusão para este trabalho e indicação de sugestões para continuidade desta pesquisa.

Capítulo

6

Conclusão e Sugestões para Trabalhos Futuros

Como destaca a literatura, o movimento em favor do software aberto caminha a passos largos. Neste sentido, o IEEE 1175 e STL, consequentemente, terão espaço já que estabelecem mecanismos para comunicação entre ferramentas distintas.

Fala-se muito sobre a "crise do software" [SOMM97, PRESS97]. Neste contexto, as Metodologias de Especificação de Software Orientadas a Objetos ocupam um papel importante ao longo do processo de desenvolvimento de aplicações. É cada vez mais comum o uso de ambientes computacionais ou auxílio de artefatos de software ou ferramentas no projeto e na construção do software propriamente dito. Há uma diversidade de MESOOs e de artefatos de software que as implementa. Como já discutimos ao longo deste trabalho, existem vários fornecedores desses ambientes CASE e cada um tem sua forma proprietária de representar as especificações.

O problema é que o reaproveitamento ou reutilização de componentes ou artefatos produzidos pelos ambientes nem sempre é possível. Há, portanto, a necessidade de transferência de semântica

entre ferramentas CASE produzidas por diferentes fornecedores. Neste contexto, há necessidade de estabelecimento de um padrão de comunicação entre ferramentas produzidas por diferentes distribuidores.

Outra questão relacionada é a escolha de qual MESOO específico deve ser usado. Em uma empresa as pessoas normalmente tornam-se especialistas em um MESOO m_1 qualquer e tendem a utilizá-lo para descrever especificações $E_{A,m1}$ independente do tipo do domínio do problema (ou cenário) sendo tratado, mesmo existindo outro MESOO m_2 mais apropriado para tratar o problema de interesse. A questão se devemos escolher m_1 ou m_2 como forma de representar o domínio de um problema (vide Figura 89), como o método de especificação "correto" é uma questão difícil de responder e encontra resistências culturais até mesmo dentro de uma só empresa.

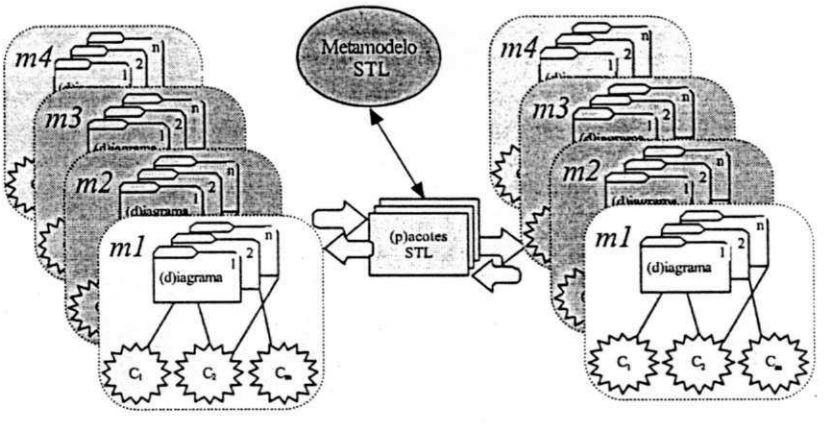

Figura 89: Escolha da MESOO "Ideal"

Sabe-se, portanto, que com a crescente demanda por software, surgem as ferramentas CASE que suportam os MESOOs M ($\{m_1, m_2, m_3, m_4\}$) e agilizam todo o processo de construção de programas. Estudar os aspectos fundamentais de cada um dos MESOOs e analisar as características das ferramentas que as suportam e fazer com que elas trabalhem de forma cooperativa é um problema que pode ser traduzido na produção de software de qualidade com maior produtividade [MEDE 94].

Contribuições do Trabalho

A principal contribuição desta tese de doutorado é a crítica e proposta de extensão do metamodelo STL para inclusão de um novo conceito chamado de *Equivalência*. Testes e implementações de artefatos de software foram elaborados e executados para ilustrar o funcionamento do interpretador de pacotes aqui proposto.

O novo metamodelo proposto vai possibilitar a comparação entre conceitos utilizados na construção de especificações E_{Am} em termos de pacotes STL. A possibilidade de registrar equivalência entre pacotes abre um leque de possibilidades para facilitar a comunicação entre Ferramentas CASE, produzidas por diferentes distribuidores, sem a necessidade de abertura dos respectivos códigos-fonte. [MEDE 2000]

Outras contribuições podem ser apontadas. Uma delas foi a implementação de um interpretador de pacotes e o ajuste da descrição BNF[15] de STL a fim de implementá-la para ser usada em

[15] Sigla para "Bakus Naur Form".

ferramentas CASE a partir de uma biblioteca de funções, como pode ser visto em maiores detalhes no Anexo A. [MEDE 2000a].

Uma outra contribuição secundária, foi o desenvolvimento de um conjunto de quatro ferramentas CASE "A4O" (Análise Orientadas a Objetos) para sentir a dificuldade de se utilizar cada um dos quatro MESOOs $m \in M$ com formatos proprietários independentes entre si. O objetivo foi construir cada uma das representações gráficas com uma forma ou tecnologia de armazenamento diferente [MEDE 2001a].

Ainda uma outra contribuição secundária foi feita ao longo de quatro semestres letivos quando tivemos oportunidade de utilizar uma abordagem de ensino de MESOOs para alunos de graduação da UFPB. Neste período, foi elaborada uma abordagem centrada na identificação dos conceitos fundamentais de cada um dos MESOOs baseados no metamodelo STL proposto, vide mais detalhes no Anexo C. [MEDE 2000b]

Podemos citar ainda como contribuição marginal uma *survey* sobre comparações entre MESOOs e experimentos realizados na disciplina de Análise e Projeto de Sistemas II no Campus I da UFPB e o desenvolvimento de um ambiente CASE Pfuncão que integra outras fases do processo de produção de software como a elicitação de requisitos. Desenvolvemos uma ferramenta CASE *EspCase* para analisar os requisitos de um programa e estimar o esforço necessário para seu desenvolvimento, como atividade complementar às ferramentas "A4O". A ferramenta *EspCase* baseia-se nas funcionalidades ou operações (Pontos por função [PRESS 97])

encontradas nas fases iniciais do processo de desenvolvimento de software, anteriores a geração dos diagramas ou construção de E_{Am} de uma aplicação a ser desenvolvida. [MEDE 2001b]

Este trabalho de tese resultou na publicação de 3 artigos técnicos no exterior. Para detalhes vide [MEDE 2000, 99, 98, 96].

Sugestões para Trabalhos Futuros

As sugestões para trabalhos futuros passam por duas vertentes: análise contínua do metamodelo STL e construção de ambientes integrados usando STL como base.

A primeira sugestão é voltada para uma análise mais aprofundada do metamodelo STL, buscando adaptar o IEEE 1175 para representar Especificações E_{Am} que utilizem novos paradigmas relacionados com maior autonomia que os componentes de software devem exigir para implementar aplicações distribuídas e tolerantes a falhas sobre a Internet. Conceitos associados com trabalho distribuído e a construção de modelos mais adequados para representar novos conceitos como agentes, comunidades, *robots*, e sistemas tolerantes a falha, devem ser endereçados.

Outras possibilidades de trabalhos futuros passam pela construção de artefatos de software para compor uma plataforma que utilize STL como forma de realizar a comunicação entre ferramentas CASE distintas, mantendo a individualidade de cada um dos componentes do ambiente.

Uma sugestão para trabalhos futuros é modificar a ferramenta *inter_stl* para gerar automaticamente repositórios de

equivalência em STL que poderiam ser utilizados em níveis hierárquicos. Neste sentido, o cálculo do custo de um software poderia ser gerado a partir de sua especificação E_{Am} em cada um dos MESOOs m, colhida em uma das quatro ferramentas "A4O$_i$", por exemplo, e utilizada diretamente pela ferramenta *EspCase*.

Partindo da Especificação de uma Aplicação E_{Am} através de uma interface de uma ferramenta como a mostrada em Figura 90 e Figura 91, a versão modificada de *inter_stl* geraria dados estatísticos e estabeleceria um repositório de equivalência hierárquico entre as diversas visões dos conceitos utilizados pelas várias ferramentas integrantes do ambiente ou plataforma de desenvolvimento STL para cálculo de estimativas associadas com os conceitos presentes em cada MESOO m.

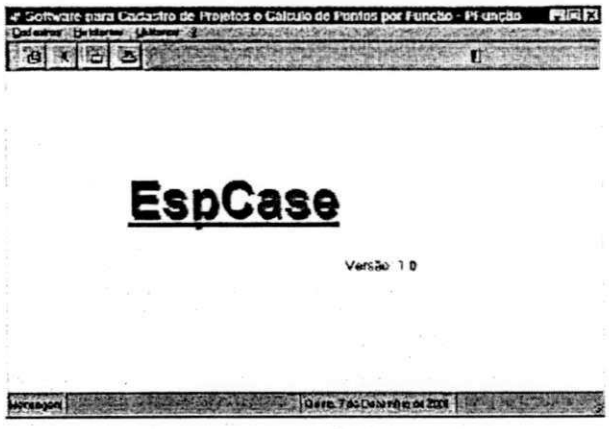

Figura 90: CASE usada para Estimar Recursos

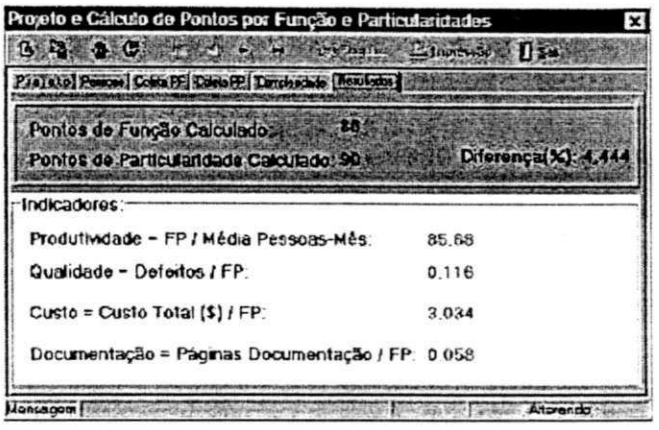

Figura 91: Cálculo da Produtividade

A seguir, apresentaremos informações sobre o ambiente operacional e mais detalhes sobre a implementação do Interpretador de Pacotes STL.

A

Anexo A: Interpretador de Pacotes STL

Neste apêndice introduziremos o Interpretador de Pacotes STL, construído com o objetivo de interpretar especificações escritas em STL conforme estabelece o Padrão IEEE 1175.

Utilizamos as ferramentas *lex* e *yacc* do Ambiente UNIX para automatizar a gramática de STL e interpretar os pacotes do cenário.

A.1. O Ambiente Operacional

Como estamos falando de portabilidade e da padronização da transferência de arquivos, vamos comentar o uso dos utilitários *lex* e o *yacc* nos Ambientes UNIX (Linux) e Windows.

O final da linha do texto fonte da descrição nestes dois ambientes operacionais é diferente. No primeiro, a linha termina com nova linha '\n' enquanto que no segundo a indicação do final da linha é delimitada por dois caracteres '\r' e '\n'.

Ao se utilizar as descrições de um ambiente em outro é importante a sinalização para o utilitário de cópia fazer a conversão de final de linha correto.

Sintaxe versus Semântica

O *lex* é o utilitário responsável pelo reconhecimento dos tokens existentes nos pacotes. O *lex* funciona integrado ao yacc. O reconhecimento sintático é feito pelo *lex* e as ações semânticas ficam sob a responsabilidade do *yacc*.

Tanto o *lex* quanto o *yacc* são utilitários que recebem arquivos com a especificação das gramáticas a serem automatizadas. Geralmente os arquivos que contêm as descrições utilizadas pelos dois comandos têm os sufixos ".l" e ".y", respectivamente.

Construção do Interpretador de Pacotes

A construção do interpretador de pacotes passou por uma adaptação da Gramática STL para o formato aceito pelos comandos *stl* e *yacc*.

Construímos a descrição *stl.l*, a partir da definição dos símbolos e tokens presentes em STL. O utilitário *lex* gerou uma rotina em C (*lex.yy.c*) para ser usada na identificação de tokens.

A Gramática STL precisou ser adaptada para a forma BNF pura, eliminando ambigüidades, para que funcionasse corretamente com o *yacc*. A gramática foi armazenada no arquivo *stl.y*. Assim como lex, o yacc gera também uma rotina na Linguagem C que necessita compilação e linkedição (comando *gcc* do Linux) para se transformar no programa *inter_stl* (nosso interpretador de pacotes). A sequência

para geração do interpretador de pacotes stl é a seguinte, partindo da linha de comandos do Ambiente Korn Shell do Linux:

```
$ yacc -d stl.y

$ lex -i stl.l

$ gcc lex.yy.c y.tab.c -o inter_stl –lgdbm
```

As especificações E_{Am} são mapeadas em pacotes stl (pacotes.stl) e interpretados por *inter_stl*.

A ativação de *inter_stl* poderá ser feita por ferramentas CASE que desejem utilizar o padrão proposto sem ter que abrir mão da confidencialidade da organização interna da estrutura de seu software. Uma biblioteca com a função *inter_stl* poderá ser oferecida para linkedição de ferramentas de fabricantes diferentes.

A.2. A Gramática STL

A visão operacional de STL apresenta, através da descrição de **pacotes** em linguagem BNF (Bacus Naur Form) modificada, detalhes de como construir ou automatizar a transferência de blocos (ou pacotes) de informação entre ferramentas CASE. A notação é composta de símbolos e a definição de pacotes.

A Tabela 7 apresenta os símbolos que serão usados para descrição dos Pacotes STL.

Tabela 7 Símbolos STL da Notação BNF

Símbolo	**Descrição do Significado**
::=	"é definido como"
Espaço, Tab, Novas Linhas	Separador de campos.
X | Y	Representa uma escolha entre *X* ou *Y*.
{X}	Zero ou mais repetições de *X*.
n{X}m	Pelo menos *n* e no máximo *m* repetições de *X*.
[X]	Indica que *X* é opcional. Ele pode ocorrer ou não.
<X>	Indica que *X* não é um elemento terminal.
X	Indica que *X* é um elemento terminal.
(Texto)	Indica que *Texto* é um comentário.
NULL	Indica uma expressão vazia.

A idéia de STL é permitir que pacotes de informações sejam trocados entre ferramentas CASE. Os pacotes são decompostos em uma ou mais sentenças STL, como mostra a regra abaixo:

<T_Packet> ::= <indentificador_sentença>
{ <sentenças_STL> }
pe_mark (Marca de Fim de pacote)

T_Packet é definido através de um identificador de sentenças seguido por uma ou mais sentenças STL. O pacote termina quando for encontrada a marca "pe_mark". O identificador de sentença "T_Packet" indica o início de um novo pacote.

As sentenças STL são construídas em função de outros elementos da linguagem tais como uma palavra-chave que determina qual conceito a ser usado, um identificador de setença seguido de uma ou mais cláusulas STL, como podemos ver a seguir:

< sentença_STL_> ::=

<keyword_sentença> <identificador_sentença>

<cláusula_STL> { ; < cláusula_STL>}.

A palavra-chave deve ser a primeira palavra em cada sentença STL. Estas estão relacionadas com os nomes dos conceitos do metamodelo utilizado (Tabela 8). Já o *identificador_setença* é um termo simbólico para identificar a sentença de forma única.

Tabela 8 Nomes de Conceitos

Action	DataItem	DataView	T_Packet
Collection	DataKey	EventItem	State
Condition	DataPart	EventType	StateTransition
ConnectionPath	DataRole	GraphicSymbol	
Constant	DataStore	Object	

As cláusulas descrevem o relacionamento entre os conceitos ou detalham um atributo presente na sentença da qual a cláusula faz

parte. Uma cláusula por ser vazia, como podemos ver a descrição abaixo que utiliza NULL com este objetivo.

<cláusula_STL> ::= <cláusula_relação> |

<cláusula_atributo> | NULL

As cláusulas de relação descrevem o relacionamento entre os conceitos da especificação de uma aplicação A escritas em uma MESOO m (E_{Am}) e o conceito STL usado na sentença da qual a cláusula é componente.

As cláusulas de atributos contêm propriedades ou características intrínsecas do conceito utilizado em E_{Am} sendo definidas na sentença STL.

A.3. Analisador Sintático

Veja a seguir exemplo de uma descrição utilizada para construção do analisador sintático a ser utilizado pelo *lex* para gerar o analisador léxico do *inter_stl* exemplo.

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

Separator    [\;\,]
Delimiter1   [\=\>\<\!\~\?\:\+\-\*\/\&\|\^\%]
```

```
HexDigit     [0-9a-fA-F]
Digit        [0-9]
OctalDigit   [0-7]
TetraDigit   [0-3]
NonZeroDigit      [1-9]
Letter       [a-zA-Z_]
AnyButSlash  [^\/]
AnyButAstr   [^\*]
BLK          [\b]
TAB          [\t]
FF           [\f]
ESCCHR       [\\]
CR           [\r]
LF           [\n]


Escape       [\\]([r]|[n]|[b]|[f]|[t]|[\\])
Identifier   {Letter}({Letter}|{Digit})*

comment1
[\/][\*]({AnyButAstr}|[\*]{AnyButSlash})*[\*][\/]
comment2          [\/][\/].*
comment            ({comment1}|{comment2})
whitespace   ({FF}|{TAB}|{BLK}|[ ])
newline            ({CR}|{LF})
```

```
AnyChrChr   [^\\']
AnyStrChr   [^\\"]
Character   [\'](({Escape}|{AnyChrChr})[\']
String      [\"](({Escape}|{AnyStrChr})*[\"]
Numeric     ({IntegerLiteral}|{FloatingPoint})
Literal          ({Numeric}|{Character}|{String})

%%
"t_packet"              {    printf("in:    T_Packet\n");
return(S_PACKET_TOKEN); }
"collection"                {                printf("in:
Collection\n"); return(COLLECTION_TOKEN); }
"datatype"              {    printf("in:    DataType\n");
return(DATATYPE_TOKEN); }
"datapart"              {    printf("in:    DataPart\n");
return(DATAPART_TOKEN); }
"has subject"               {       printf("in:       has
subject\n"); return(HAS_SUBJECT_TOKEN); }
"has stl version"           {   printf("in:   has   stl
version\n"); return(HAS_STL_VERSION_TOKEN); }
"has content version"       { printf("in: has content
version\n"); return(HAS_CONTENT_VERSION_TOKEN); }
"has content timestamp"          {    printf("in:  has
content                                  timestamp\n");
return(HAS_CONTENT_TIMESTAMP_TOKEN); }
```

```
"has component datapart"    { printf("in: has component
datapart\n"); return(HAS_COMPONENT_DATAPART_TOKEN); }

"has creator"               {       printf("in:       has
creator\n"); return(HAS_CREATOR_TOKEN); }

"has originating tool"          {    printf("in:    has
originating                                    tool\n");
return(HAS_ORIGINATING_TOOL_TOKEN); }

"has label"           {   printf("in:   has   label\n");
return(HAS_LABEL_TOKEN); }

"has description"           {       printf("in:       has
description\n"); return(HAS_DESCRIPTION_TOKEN); }

"has units"           {   printf("in:   has   units\n");
return(HAS_UNITS_TOKEN); }

"has format"                {       printf("in:       has
format\n"); return(HAS_FORMAT_TOKEN); }

"has standard usage"        { printf("in: has standard
usage\n"); return(HAS_STANDARD_USAGE_TOKEN); }

"has external usage"        { printf("in: has external
usage\n"); return(HAS_EXTERNAL_USAGE_TOKEN); }

"is datatypeclass"          {       printf("in:       is
datatypeclass\n"); return(IS_DATATYPECLASS_TOKEN); }

"is specified as datatype"  { printf("in: is specified
as                                        datatype\n");
return(IS_SPECIFIED_AS_DATATYPE_TOKEN); }

"groups datatype"           {     printf("in:     groups
datatype\n"); return(GROUPS_DATATYPE_TOKEN); }
```

```
"is grouped into collection"     {     printf("in:     is
grouped                into                collection\n");
return(IS_GROUPED_INTO_COLLECTION_TOKEN); }

(dfd|erd|std|designunit)    {     printf("in:     standard
usage\n"); return(STANDARD_USAGE); }

(unspecified|integer|real|character|string|boolean|se
quence|choice|array|table|relationship|entity) {

                                printf("in:
DataTypeClass\n"); return(DATATYPECLASS_TOKEN);

                        }

"pe_mark"               {   printf("in:    end    of    stl
file\n"); return(PE_MARK); }


{String}                {       printf("in:        string:
%s\n",yytext); return(STRING_LITERAL); }

{Identifier}                {

                          yylval.str=(char
*)strdup(yytext);

                          return (IDENTIFIER);

                        }


{Separator}             { return(yytext[0]); }


{newline}               { }

{whitespace}                { }

{comment}               { }
```

```
%%

yywrap() {
  return(1);
}

comment() {
  char c, c1;
  loop:
  while ((c = input()) != '*' && c != 0)
    putchar(c);
    if ((c1 = input()) != '/' && c != 0)    {
      unput(c1);
      goto loop;
    }
    if (c != 0)
      putchar(c1);
}
```

A partir desta especificação, o utilitário *lex* gera uma função na Linguagem C a ser utilizada junto com outra função gerada pelo *yacc* para implementar o interpretador de pacotes stl exemplo.

A.4. Análise Semântica

Veja a seguir exemplo de uma descrição utilizada para construção do analisador semântico a ser utilizado pelo *yacc* para gerar o analisador léxico do *inter_stl*/exemplo.

```
%union {
  char *str;
  int num;
}
%token <str> IDENTIFIER
%token S_PACKET_TOKEN
%token COLLECTION_TOKEN
%token DATATYPE_TOKEN
%token DATAPART_TOKEN
%token DATATYPECLASS_TOKEN
%token IS_DATATYPECLASS_TOKEN
%token IS_SPECIFIED_AS_DATATYPE_TOKEN
%token IS_GROUPED_INTO_COLLECTION_TOKEN
%token HAS_SUBJECT_TOKEN
%token HAS_STL_VERSION_TOKEN
%token HAS_CONTENT_VERSION_TOKEN
%token HAS_CONTENT_TIMESTAMP_TOKEN
%token HAS_CREATOR_TOKEN
%token HAS_ORIGINATING_TOOL_TOKEN
%token HAS_LABEL_TOKEN
%token HAS_DESCRIPTION_TOKEN
%token HAS_STANDARD_USAGE_TOKEN
%token HAS_COMPONENT_DATAPART_TOKEN
%token STANDARD_USAGE
%token HAS_EXTERNAL_USAGE_TOKEN
%token HAS_FORMAT_TOKEN
```

```
%token HAS_UNITS_TOKEN
%token GROUPS_DATATYPE_TOKEN
%token STRING_LITERAL
%token PE_MARK
%start stl_file
%%
stl_file                : S_Packet PE_MARK
                        ;
S_Packet                : S_PACKET_TOKEN IDENTIFIER
s_packet_attributes s_packet_components
                        { s_packet+=store_symbol($2,"S_Packet");
printf("S_Packet -> %s\n",$2); }
                        ;
s_packet_attributes     : s_packet_attribute ';'
                | s_packet_attribute ';' s_packet_attributes
                        ;
s_packet_attribute      : /* no attributes */
                        | has_subject
                        | has_stl_version
                        | has_content_version
                        | has_content_timestamp
                        | has_creator
                        | has_originating_tool
                        | has_label
                        | has_description
                        ;
s_packet_components     : s_packet_component
                | s_packet_component s_packet_components
                        ;
s_packet_component      : Collection
                        | DataType
                        | DataPart
                        ;
```

```
Collection : COLLECTION_TOKEN IDENTIFIER
collection_attributes collection_relations
                { resolve_relations($2);
collection+=store_symbol($2,"Collection"); printf("Collection -> %s\n",$2); }
                ;
DataType : DATATYPE_TOKEN IDENTIFIER
datatype_attributes
                { resolve_relations($2);
datatype+=store_symbol($2,"DataType"); printf("DataType -> %s\n",$2); }
                ;
DataPart : DATAPART_TOKEN IDENTIFIER
datapart_attributes
                { resolve_relations($2);
datapart+=store_symbol($2,"DataPart"); printf("DataPart -> %s\n",$2); }
                ;
collection_attributes : collection_attribute ';'
        | collection_attribute ';' collection_attributes
                ;

datatype_attributes : datatype_attribute ';'
        | datatype_attribute ';' datatype_attributes
                ;
datapart_attributes : datapart_attribute ';'
        | datapart_attribute ';' datapart_attributes
                ;
collection_attribute : has_label
                | has_description
                | has_standard_usage
                | has_external_usage
                ;
datatype_attribute : has_label
                | has_description
                | has_format
                | has_units
                | is_datatypeclass
```

```
                        | is_grouped_into
                        | has_component_datapart
                        ;

datapart_attribute      : has_description
                        | is_specified_datatype
                        ;
collection_relations    : collection_relation ';'
                        | collection_relation ';' collection_relations
                        ;

collection_relation     : groups_datatype
                        ;
groups_datatype         : GROUPS_DATATYPE_TOKEN DataTypes
        ;
has_component_datapart  :  HAS_COMPONENT_DATAPART_TOKEN
DataParts
                        ;

DataTypes               : DataTypeId
                        | DataTypeId ',' DataTypes
                        ;

DataTypeId              : IDENTIFIER
                        { push_relation("DataType",$1); }
                        ;

DataParts               : DataPartId
                        | DataPartId ',' DataParts
                        ;

DataPartId              : IDENTIFIER
                        { push_relation("DataPart",$1); }
                        ;
```

```
is_grouped_into        :    IS_GROUPED_INTO_COLLECTION_TOKEN
IDENTIFIER             { push_relation("Collection",$2); }

        ;

has_subject                    : HAS_SUBJECT_TOKEN string
                       { printf("Subject -> OK\n"); }

                               ;

is_datatypeclass       :                    IS_DATATYPECLASS_TOKEN
DATATYPECLASS_TOKEN            { printf("DataTypeClass -> OK\n"); }

                               ;


is_specified_datatype  :          IS_SPECIFIED_AS_DATATYPE_TOKEN
DATATYPECLASS_TOKEN    { printf("DataPart Type -> OK\n"); }

                               ;


has_stl_version        : HAS_STL_VERSION_TOKEN string
                  { printf("STL Version -> OK\n"); }

                               ;


has_content_version    : HAS_CONTENT_VERSION_TOKEN string
                       { printf("Content Version -> OK\n"); }

                               ;


has_content_timestamp  : HAS_CONTENT_TIMESTAMP_TOKEN string
                  { printf("Content Timestamp -> OK\n"); }

                               ;


has_creator                    : HAS_CREATOR_TOKEN string
                       { printf("Creator -> OK\n"); }

                               ;


has_originating_tool   : HAS_ORIGINATING_TOOL_TOKEN string
                  { printf("Originating Tool -> OK\n"); }

                               ;
```

```
has_label               : HAS_LABEL_TOKEN string
                        { printf("Label -> OK\n"); }
                        ;

has_description         : HAS_DESCRIPTION_TOKEN string
                        { printf("Description -> OK\n"); }
                        ;

has_standard_usage      :           HAS_STANDARD_USAGE_TOKEN
STANDARD_USAGE                      { printf("Standard Usage -> OK\n"); }
                        ;

has_external_usage      : HAS_EXTERNAL_USAGE_TOKEN string
                        { printf("External Usage -> OK\n"); }
                        ;

has_format              : HAS_FORMAT_TOKEN string
                        { printf("Format -> OK\n"); }
                        ;

has_units               : HAS_UNITS_TOKEN string
                        { printf("Units -> OK\n"); }
                        ;

string                  : STRING_LITERAL
                        ;

%%
#include <stdio.h>
#include <string.h>
#include <gdbm.h>

GDBM_FILE dbm_symbol;
extern char yytext[];
extern int yylineno;
```

```
int collection;
int s_packet;
int datatype;
int datapart;
int relations_count;
int undefined_count;

char actual_symbol[128];
char relations[64][128];
char undefined[64][128];

void yyerror(char *s) {
 fflush(stdout);
 printf("Syntax Error!\n");
}

int main(int argc, char **argv) {
 dbm_symbol=gdbm_open("symbol.db", 512, GDBM_NEWDB, 0644, NULL);
 relations_count=0;
 undefined_count=0;
 yyparse();
 printf("========================                    T_PACKET
==========================\n");
 if (s_packet>0) { printf ("%5d S_Packet \n",s_packet);
list_symbols("S_Packet"); }
 if (collection>0) { printf ("%5d Collection \n",collection);
list_symbols("Collection"); }
 if (datatype>0) { printf ("%5d DataTypes \n",datatype);
list_symbols("DataType"); }
 if (datapart>0) { printf ("%5d DataPart \n",datapart);
list_symbols("DataPart"); }

printf("==========================================
==============\n");
```

```
 if (undefined_count>0) { printf ("%5d Undefined
Objects:\n",undefined_count); list_undefined(); }
 gdbm_close(dbm_symbol);
 return 0;
}

int store_symbol (char* symbol, char* symbol_type) {
 int c;
 for (c=0;c<strlen(symbol);c++) actual_symbol[c]=symbol[c];
actual_symbol[c]='\0';
 if (gdbmstore(symbol, symbol_type)==0) return 1;
 return 0;
}

void push_relation (char* relation_type, char* relation) {
 if (relations_count<64) {
   sprintf(relations[relations_count],"%s %s",relation_type,relation);
   relations_count++;
 }
 return;
}

void resolve_relations (char* entity) {
 char temp[128];
 int c;
 for (c=0; c<relations_count; c++) {
  sprintf(temp,"%05d%s",c,entity);
  gdbmstore(temp,relations[c]);
 }
 relations_count=0;
}

int list_undefined (void) {
```

```
 int c;
 for (c=0; c<undefined_count; c++) {
  printf("\t\t\t%s\n", undefined[c]);
 }
 return 0;
}

int list_symbols(char* symbol_type) {
 char str[128]; int c;
 datum key, data;
 key=gdbm_firstkey(dbm_symbol);
 while (key.dptr!=NULL) {
   data=gdbm_fetch(dbm_symbol, key);
   if (data.dptr!=NULL) {
            for (c=0;c<data.dsize;c++) str[c]=data.dptr[c]; str[c]='\0';
            // printf("seeking on symbol: %s (%s)\n", symbol_type, str);
            if (strcmp(symbol_type,str)==0) {
              for (c=0;c<key.dsize;c++) str[c]=key.dptr[c]; str[c]='\0';
        printf ("\t\t\t%s\n", str);
              list_relations(str);
            }
       free (data);
   }
   key=gdbm_nextkey(dbm_symbol,key);
 }
 free (key);
}

int list_relations(char* entity) {
 char temp[128]; char str[128];
 int found=1;
 int c=0; int cx;
 datum key, data;
```

```
 while (found) {
  sprintf(temp,"%05d%s",c,entity);
  key.dptr=temp; key.dsize=strlen(key.dptr);
  data=gdbm_fetch(dbm_symbol, key);
  if (data.dptr!=NULL) {
   for (cx=0;cx<data.dsize;cx++) str[cx]=data.dptr[cx]; str[cx]='\0';
   printf("\t\t\t - %s ",str);
   verify(str);
   c++;
  } else {
   found=0;
  }
 }
}

int verify (char* coded) {
 char entity_type[128];
 char entity[128];
 char str[128];
 int c;
 datum key, data;
 sscanf(coded, "%s %s", entity_type, entity);
 key.dptr=entity; key.dsize=strlen(key.dptr);
 data=gdbm_fetch(dbm_symbol, key);
 if (data.dptr==NULL) {
  printf("(not defined!)\n");
  sprintf(undefined[undefined_count],"%s %s",entity_type, entity);
  undefined_count++;
 } else {
  printf("\n");
 }
 return 0;
}
```

```
int gdbmstore(char* keytxt, char* datatxt) {
  datum key, data;
  key.dptr=keytxt; key.dsize=strlen(key.dptr);
  data.dptr=datatxt; data.dsize=strlen(data.dptr);
  return gdbm_store(dbm_symbol, key, data, GDBM_INSERT);
}
```

A seguir, apresentaremos uma visão de geral dos conceitos de cada um dos MESOOs baseada nas cláusulas STL.

Anexo

B

Anexo B: Visão Geral dos Conceitos das MESOOs Baseadas nas Cláusulas STL

Analisando cada uma das metodologias de especificação de software orientadas a objetos (MESOOs) em termos de suas características STL suportadas (ou não), chegamos à Tabela 9. Observe que para extrair alguma informação desta tabela é importante conhecer STL. Através da presença ou não de algumas cláusulas pode se concluir que uma determinada MESOO é mais voltada para modelar características comportamentais ou aspectos dinâmicos dos sistemas ao invés de detalhar mais seus aspectos funcionais.

B.1. Comparação de Cláusulas STL (Função das MESOOs)

Nossa pretensão não é construir uma comparação completa [MEDE 98]. Nossa proposta é que se ampliem os conceitos de STL para servir também ao propósito de representação canônica dos conceitos das ferramentas de modelagem conceitual das MESOOs. Faz-se necessário,

portanto, o agrupamento dos conceitos em STL para uma melhor representação das MESOOs em função das cláusulas STL vistas como a população de uma visão de banco de dados dos conceitos utilizados nos pacotes STL para representar as E_{Am} (Especificação E para uma Aplicação A através de uma MESOO $m \in M$), por exemplo.

Tabela 9: Visão Geral dos Métodos em Função dos Conceitos STL

Conceito ou subtipo	**Descrição**	**AOO**	**UML**	**Shlaer/Mellor**	**Wirfs-Brock**
Conceito: **AÇÃO**	Transforma ações internas ou externas; tem entrada e saída (dispositivo).				
Action	Has description desc_string	S	S	S	S
Action	Has label lab_string	S	S	S	S
Action	Has component action ActionID {, ActionID}	S	S	S	S
Action	is connected from connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
Action	is connected to connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
Action	Produces dataitem DataItemID {, DataItemID}	S	S	S	S
Action	Receives eventitem EventItem {, EventItem}	S	S	S	S
Action	Transmits eventitem EventItem {, EventItem}	S	S	S	S
Action	Uses dataitem DataItemID {, DataItemID}	S	S	S	N
External	Is actiontype external	N	S	S	S
External	Has occurence solicited	N	S	N	S
Internal	Has criticality mandatory | desirable | op tional	N	S	N	N
Internal	Has time units time_units	N	S	S	N
Internal	Has transform purpose control | data | compound	S	S	S	S
Internal	is action type internal	S	S	N	S
Internal	Has execution time e_time	N	N	S	N
Internal	Causes statetransition StateTransit ionID {, StateTransitionID}	S	S	S	S
Internal	is allowed in state StateID {, StateID}	S	S	S	N
Internal	is encapsulated in object ObjectID {, ObjectID}	S	S	S	S
Conceito: **DISPOSITIVO**	Fornece entrada e saída para elementos do sistema (eventos).				
Internal	is offered by object ObjectID {, ObjectID}	S	S	S	S
Internal	Satisfy condition ConditionID {, ConditionID}	S	S	S	S
Internal	Creates DataItemID {, DataItemID}	N	S	S	N
Internal	Reads DataItemID {, DataItemID}	N	S	S	N
Internal	writes DataItemID {, Dat aItemID}	N	S	S	N
Internal	deletes DataItemID {, DataItemID}	N	S	S	N
ConnectionPath	has label lab_estring	S	S	S	S
ConnectionPath	has flow characteristics conditional | uncond.	S	S	S	N
ConnectionPath	connects from action ActionID	N	S	S	S

ConnectionPat h	connects to action ActionId	N	S	S	S
ConnectionPath	has component connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
ConnectionPath	is bidirectional with connectionpath ConnectionPathID	S	S	S	
ConnectionPath	is encapsulated in object ObjectID {, ObjectID}	S	S	S	N
ConnectionPath	is merged from connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
ConnectionPath	merges into connectionpath ConnectionPathID	S	S	S	S
ConnectionPath	splits into connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
Data	has flow characteristics discrete | continuous	N	N	N	N
Data	has retrieval effect nondestructive | destructive	N	N	N	N
Data	is connectiontype data	S	S	S	S
Data	carries dataitem DataItemID {, DataItemID}	S	S	S	N
Data	connects from datastore DataStoreID	N	N	S	N
Data	connects to datastore DataStoreID	N	N	S	N
Event	is connectiontype event	S	S	S	S
Event	carries eventitem EventItemID {,eventItemID}	S	S	S	S
Conceito: **DADOS**	Uma instância que serve como uma entrada ou uma saida de uma ação ou dispositivo				
DataItem	has description desc_string	S	S	S	S
DataItem	has label lab_string	S	S	S	S
DataItem	has format dat_format	S	S	S	S
DataItem	has null occurrences allowed | disallowed	S	S	S	S
DataItem	has units dat_units	N	N	S	N
DataItem	is accepted by datastore DataStoreID {, DataStoreID}	N	N	S	N
DataItem	is carried by connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
DataItem	is characterized by state StateID {, StateID}	S	S	N	N
DataItem	is encapsulated in object ObjectID {, ObjectID}	S	S	S	S
DataItem	is offered by object ObjectID {, ObjectID}	S	S	S	S
DataItem	is produced by action ActionID {, ActionID}	S	S	S	S
DataItem	is stored by datastore DataStoreID {, DataStoreID}	N	S	S	N
DataItem	is supplied by datastore DataStoreID {, DataStoreID}	N	S	S	N
DataItem	is used by action ActionID {, ActionID}	S	S	S	S
DataKey	Uma referência para uma ou mais *dataparts* cujos valores servem para identificar unicamente as diferentes instâncias de *datatype*				
Internal	has description desc_string	N	S	N	N
Internal	has label lab_string	N	S	N	N
Internal	has identify purposes primary | alternate | foreign	S	S	N	N
Internal	references datapart DataPartID {, DataPartID}	S	S	S	S
Internal	shares instances with datakey DataKeyID {, DataKey ID}	S	S	S	S
DataPart	Um componente de uma estrutura *datatype* com o domínio referenciando outro tipo				
DataPart	has description desc_string	S	S	S	S
DataPart	has label lab_string	S	S	S	S
DataPart	has format dat_format	S	S	S	N
DataPart	has null occurrences allowed | disallowed	S	S	S	N
DataPart	has units dat_units	N	N	S	N
DataPart	is referenced by datakey DataKeyID {, DataKeyID}	S	S	S	S
DataRole	Caracteriza a forma como uma entidade *datatype* participa de um relacionamento				

DataRole	has description desc_string	S	S	S	S
DataRole	has label lab_string	S	S	S	S
DataRole	has expression exp_text	N	S	N	S
DataRole	has inner cardinality maximum max_cardinality	N	S	N	N
DataRole	has inner cardinality minimum min_cardinality	N	S	N	N
DataRole	has inner cardinality maximum unbounded	N	S	N	N
DataRole	has outer cardinality maximum max_cardinality	N	S	N	N
DataRole	has outer cardinality minimum min_cardinality	N	S	N	N
DataRole	has outer cardinality maximum unbounded	N	S	N	N
DataRole	has component datapart Dat aPartID {, DataPartID}	N	S	S	N
DataStore	Especificação dos dados necessários				
DataStore	Has description desc_string	N	N	N	N
DataStore	Has label lab_string	N	N	N	N
DataStore	Has retrieval effect depletable | nondepletable	N	N	N	N
DataStore	Accepts dataitem DataItemID {, DataItemID}	N	N	N	N
DataStore	Has component datastore DataStoreID {, DataStoreID}	N	N	N	N
Internal	is connected from connectionpath ConnectionPathID {, ConnectionPathID}	S	S	S	S
Internal	is connected to connectionpath Connec tionPathID {, ConnectionPathID}	S	S	S	S
DataStore	stores dataitem DataItemID {, DataItemID}	S	S	S	N
DataStore	supplies dataitem DataItemID {, DataItemID}	S	S	S	N
DataType	Um conjunto de possíveis valores ou estruturas que um *dataitem* pode assumir				
DataType	has description desc_string	S	S	S	S
DataType	has label lab_string	S	S	S	S
DataType	has format dat_format	S	S	N	S
DataType	has units dat_units	N	N	N	N
DataType	instantiates dataitem DataItemID {, DataItemID}	S	S	N	S
DataType	Specify datapart DataPartID {, DataPartID}	S	S	S	S
Entity	is datatypeclass entity	S	S	S	S
Entity	has component datapart DataPartID {, DataPartID}	S	S	S	S
Entity	is subtype in dataview DataViewID {, DataViewID}	S	S	S	S
Entity	is accessed through datakey DataKeyID { , DataKeyID}	S	S	S	S
Entity	is particioned by dataview DataViewID {, DataViewID}	S	S	S	S
Entity	plays datarole DataRoleID {, DataRoleID}	S	S	S	S
Relationship	is datatypeclass relationship	S	S	S	S
Relationship	has component datapart DataPartID {DataPart ID}	S	S	S	S
Relationship	involves datarole DataRoleID {, DataRoleID}	S	S	S	S
Conceito: **Evento**	Uma instância de um evento usado em uma entrada ou em uma saída (dispositivo) de uma *action*, representa um possível ponto de sincronização de ações				
EventItem	has description desc_string	N	S	S	N
EventItem	has label lab_string	N	S	S	N
EventItem	is carried by connectionpath ConnectionPathID {, ConnectionPathID}	N	S	S	N
EventItem	is encapsulated in object ObjectID {, ObjectID}	S	S	S	S
EventItem	is offered by object ObjectID {, ObjectID}	S	S	S	S
EventItem	is received by action ActionID {, ActionID}	S	S	S	N
EventItem	is transmitted by action ActionID {, ActionID}	S	S	S	N
EventItem	triggers on clock clock_name	N	S	S	S

EventItem	resets clock clock_name	N	S	S	N
EventItem	triggers on clock time clock_time	N	S	S	N
EventItem	detects action ActionID	S	S	S	N
EventItem	goes to object ObjectID	S	S	S	N
EventItem	carries supplemental data DataItemID {, DataItemID}	N	S	S	N
EventItem	carries datakey DataKeyID	N	S	S	N
EventStore	Um mecanismo para armazenar informações sobre eventos relacionando dados históricos				
EventStore	has description desc_string	N	S	S	N
EventStore	has label lab_string	N	S	S	N
EventStore	has retrieval effect depletable | non depletable	N	S	S	N
EventStore	accepts eventitem EventItemID {, EventiItemID}	N	S	S	N
EventStore	stores eventitem EventItemID {, EventiItemID}	N	S	S	N
EventStore	supplies eventitem EventItemID {, EventiItemID}	N	S	S	N
EventType	Um cojunto de possiveis ocorrências ou periodos em que um *eventitem* pode ocorrer				
EventType	has description desc_string	N	S	S	N
EventType	has label lab_string	N	S	S	N
EventType	has action effect trigger enable | disable	N	S	S	N
EventType	Instanciates eventitem EventItemID { , EventItemID}	N	S	S	N
Coincident	is eventtypeclass coincident	N	S	S	N
Coincident	has component eventitem EventItemID {, EventItemID}	N	S	S	N
Consequent	is eventtypeclass consequent	N	S	S	N
Consequent	has component eventitem EventItemID {, EventItemID}	N	S	S	N
Unspecified	is eventtypeclass unspecified	S	S	S	N
Unstructured	is eventtypeclass unstructured	S	S	S	N
State	A instância de uma estrutura simples ou agrupada				
State	has description desc_string	N	S	S	N
State	has label lab_string	N	S	S	N
State	has purpose initial | intermediate | final	N	S	S	N
State	has time units time_units	N	S	S	N
State	has dwell time dw_time	N	S	S	N
State	is encapsulated in object ObjectID {, ObjectID}	N	S	S	S
Action	is statetype action	N	S	S	S
Action	allows action ActionID {, ActionID}	N	S	S	S
Concurrent	is statetype concurrent	N	S	N	N
Data	is statetype data	N	S	S	S
Data	characterizes dataitem DataItemID {, DataItemID}	N	N	N	S
Data	is stablished by condition ConditionID	N	S	S	N
Sequencial	is statetype sequential	N	S	S	N
Sequencial	has component state StateID {, StateID}	N	S	S	N
THREADS	Um conjunto de estados que representam uma possivel seüência de eventos ou ações				
StateTransition	has description desc_string	N	S	S	N
StateTransition	Has label lab_string	N	S	S	N
StateTransition	is caused by action ActionID {, ActionID}	N	S	S	N
StateTransition	is encapsulated in object ObjectID {, ObjectID}	N	S	S	S
Action	is statetransitiontype action	N	S	N	S
Action	Goes to first state StateID	N	S	S	N
Action	Goes to second state StateID	N	N	N	N
Compound	is statetransitiontype compound	N	S	S	N
Compound	Has component statetransition StateTransitionID {,	N	S	S	N

	StateTransitionID}				
Data	is statetransitiontype data	S	S	N	N
Data	Goes from first state StateID	N	S	S	N
Data	Goes to second state StateID	N	S	S	N
Status	is statetransitiontype status	N	S	S	N
Status	is status ongoing continous event flow	N	S	S	N
Status	is status clock interval	N	S	S	N
Status	is status union continous event flow	N	N	S	N
Status	Is status not continous event flow	N	S	S	N
Status	Is status either ConditionID {, ConditionID}	N	S	N	N
Status	Is status available continous event flow	N	N	N	N

Tabela 10: Representando Ferramentas de Modelagem das MESOOs

Conceito STL	**Conceito STL**	**Descrição da Relação STL**	**AOO**	**UML**	**SHAL**	**WOOD**
Action	Action	Is component of action ActionID{, ActionID}	X	X	X	X
Action	Action	Is grouped into collection CollectionID{,CollectionID}	-	-	-	X
Action	Action	Produces dataitem DataItemID{,DataItemID}	X	X	X	X
Action	Action	Receives eventitem EventItemID{,EventItemID}	X	-	X	X
Action	Action	Trasmits eventitem EventItemID{,EventItemID}	X	-	X	X
Action	Action	Uses dataitem DataItemID{,DataItemID}	X	X	X	X
Action	External	Is actiontype external	-	-	-	-
Action	Internal	Acts only if condition ConditionID	X	-	X	-
Action	Internal	Causes statetransition StateTransID{,StateTransID}	X	-	X	-
Action	Internal	Is allowed in state StateID{,StateID}	X	-	X	-
Action	Internal	Satisfies condition ConditionID	N [16]	-	N	-
Collection	Collection	Groups action ActionID{,ActionID}	N	-	-	X
Collection	Collection	Groups condition ConditionID{,ConditionID}	N	-	-	-
Collection	Collection	Groups dataitem DataItemID{,DataItemID}	N	-	-	X
Collection	Collection	Groups datahole DataHoleID{,DataHoleID}	N	-	-	X
Collection	Collection	Groups datastore DataStoreID{,DataStoreID}	N	-	-	N
Collection	Collection	Groups datatype DataTypeID{,DataTypeID}	N	-	X	X
Collection	Collection	Groups dataview DataViewID{,DataViewID}	N	-	-	-
Collection	Collection	Groups eventitem EventItemID{,EventItemID}	N	-	X	X
Collection	Collection	Groups eventype EventTypeID{,EventTypeID}	N	-	X	X
Collection	Collection	Groups state StateID {,StateID}	N	-	-	-
Collection	Collection	Groups statetransition StateTransID {,StateTransID}	N	-	-	-
Collection	Collection	Is component of collection CollectionID{,CollectionID}	N	-	X	X
Condition	Condition	Establishes state StateID {,StateID}	X	X	X	-
Condition	Condition	Is component of condition ConditionID{,ConditionID}	-	X	-	-

[16] "N" significa não é aplicável; "-" significa não existente; "X" indica presença na MESOO.

Condition	Condition	Is grouped into collection CollectionID {,CollectionID}	-	-	-	-
Condition	Condition	Is satisfied by action ActionID {,ActionID}	-	X	X	-
Condition	Condition	Permits action ActionID {,ActionID}	X	X	X	-
DataItem	DataItem	Is accepted by datastore DataStoreID{,DataStoreID}	X	X	X	N
DataItem	DataItem	Is an instance of constant ConstantID	-	X	N	N
DataItem	DataItem	Is characterized by state StateID {,StateID}	-	-	-	-
DataItem	DataItem	Is grouped into collection CollectionID{,CollectionID}	N	N	-	X
DataItem	DataItem	Is produced by action ActionID {,ActionID}	X	X	X	X
EventType	EventType	Instanciates eventitem EventItemID {,EventItemID}	X	X	X	X
EventType	EventType	Is grouped into collection CollectionID {,CollectionID}	N	X	X	X
State	State	Is component of state StateID {,StateID}	X	X	X	-
State	Action	Allows action ActionID {,ActionID}	X	X	X	-
StateTransitio n	StateTransitio n	Is caused by action ActionID {,ActionID}	X	X	X	-

A seguir, apresentaremos detalhes de implementação de artefatos de software desenvolvidos para implementar cada um dos MESOOs em um formato proprietário.

C

Anexo C: Ferramentas CASE

Este anexo apresenta os módulos principais do código fonte de uma ferramenta CASE que suporta uma MESOO *m* (UML) objeto desta tese. Por questão de limitação de espaço nesta dissertação, vamos publicar o fonte completo deste programa *online*. Outras três Ferramentas CASE "AOO" objeto desta tese serão também apresentadas na Web no endereço http://www.alvaro.di.ufpb.br/alvaro/fontes/CaseA4O.

Nosso objetivo é apresentar o esboço que sirva de base para construção de aplicações práticas utilizando ferramentas que utilizam formatos internos diferentes e que podem se comunicar usando com o *inter_stl*.

Esperamos que trabalhos futuros possam materializar o emprego do Padrão IEEE 1175 para comparar pacotes STL através da população de uma visão de Banco de Dados dos Conceitos STL[MEDE 96, 98]. Esperamos que nossa extensão proposta [MEDE 2000] sirva para aumentar a produtividade no desenvolvimento de software e reduzir a sua atual crise[SOMM 97].

C.1. Código Parcial Ferramenta A4O para UML

Veja a seguir cópia do código fonte da ferramenta A4O_UML usada para ilustrar uma das sugestões de trabalhos futuros.

```
unit UPrincipal;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, ToolWin, ComCtrls, ImgList, UUseCase, extctrls, Grids, DBGrids,
  Db, DBTables, UClasses, stdctrls, UObjetos;

type
  TTextos = class(TLabel)
  private
  public
    codTexto: integer;
  end;
  TClasses = class(TImage)
  private
  public
    codClasse: integer;
  end;
  TObjetos = class(TClasses)
  private
  public
    codObjeto: integer;
  end;

  TAtor = class(TImage)
  private
  public
```

```
  codAtor: integer;
end;

TUseCase = class (TImage)
private
public
  codUCase: integer;
end;

TFPrincipal = class(TForm)
  MainMenu1: TMainMenu;
  Arquivo1: TMenuItem;
  NovoProjeto1: TMenuItem;
  AbrirProjeto1: TMenuItem;
  N1: TMenuItem;
  Sair1: TMenuItem;
  Ajuda1: TMenuItem;
  Tpicos1: TMenuItem;
  Sobre1: TMenuItem;
  ToolBar1: TToolBar;
  Salvar1: TMenuItem;
  tbNovoProjeto: TToolButton;
  tbAbrirProjeto: TToolButton;
  tbSalvarProjeto: TToolButton;
  ImageList1: TImageList;
  ImageList2: TImageList;
  Exibir1: TMenuItem;
  Viso2: TMenuItem;
  UseCase2: TMenuItem;
  VisoLgica1: TMenuItem;
  Componentes1: TMenuItem;
  Concorrncia1: TMenuItem;
  Organizao1: TMenuItem;
```

```
NovoDiagrama1: TMenuItem;
UseCase1: TMenuItem;
Classes2: TMenuItem;
Objetos2: TMenuItem;
Estado2: TMenuItem;
Sequncia2: TMenuItem;
Colaborao2: TMenuItem;
Atividade2: TMenuItem;
Execuo2: TMenuItem;
Diagramas1: TMenuItem;
FecharProjeto1: TMenuItem;
N2: TMenuItem;
quDiag: TQuery;
Panel1: TPanel;
PageControl1: TPageControl;
tbUseCase: TTabSheet;
ToolBar2: TToolBar;
StatusBar1: TStatusBar;
tbAtor: TToolButton;
tbUCase: TToolButton;
DeletaProjeto1: TMenuItem;
tbFecharProjeto: TToolButton;
tsEstado: TTabSheet;
tsSequencia: TTabSheet;
tsColaboracao: TTabSheet;
tsAtividade: TTabSheet;
tsComponente: TTabSheet;
tsExecucao: TTabSheet;
ToolBar5: TToolBar;
ToolButton3: TToolButton;
ToolBar6: TToolBar;
ToolButton4: TToolButton;
ToolBar7: TToolBar;
```

```
    ToolButton7: TToolButton;
    ToolBar8: TToolBar;
    ToolButton8: TToolButton;
    ToolBar9: TToolBar;
    ToolButton9: TToolButton;
    ToolBar10: TToolBar;
    ToolButton10: TToolButton;
    tsObjetos: TTabSheet;
    ToolBar4: TToolBar;
    tbClasse: TToolButton;
    ToolButton1: TToolButton;
    MenuTexto: TPopupMenu;
    Editar1: TMenuItem;
    Apagar1: TMenuItem;
    tbObjeto: TToolButton;
    tsClasse: TTabSheet;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Sair1Click(Sender: TObject);
    procedure UseCase1Click(Sender: TObject);
    procedure tbAtorClick(Sender: TObject);
    procedure ImagemMouseDown(Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
    procedure tbUCaseClick(Sender: TObject);
    procedure NovoProjeto1Click(Sender: TObject);
    procedure AbrirProjeto1Click(Sender: TObject);
    procedure FecharProjeto1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure DeletaProjeto1Click(Sender: TObject);
    procedure Classes2Click(Sender: TObject);
    procedure tbClasseClick(Sender: TObject);
    procedure ToolButton1Click(Sender: TObject);
    procedure Apagar1Click(Sender: TObject);
    procedure Editar1Click(Sender: TObject);
```

```
    procedure tbObjetoClick(Sender: TObject);
    procedure Objetos2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    UseCase: TFUseCase; // diagrama Use Case
    Classes: TFClasses; // diagrama de Classes
    Objetos: TFObjetos; // diagrama de Objetos

    ator: TAtor;      // vai representar os atores do diagrama de use case
    uCase: TUseCase;  // vai representar os use cases
    classe: TClasses;
    objeto: TObjetos;
    textos: TTextos;

    imagem : TImage;  // vai receber as figuras dos diagramas em diferentes momentos
    texto: TLabel;

    codFiguraAux,
    codTextoAux,
    codProjeto,
    posX,
    posY: integer;

    tipoDiag: string;

    criou: boolean;  // usada para verificar se foi criado um diagrama
  end;
var
  FPrincipal: TFPrincipal;
implementation
uses UNovoProj, UDataModule1, UAbreProj, UNovoDiag, UChildForm, UEditTexto;
```

```
{$R *.DFM}
procedure TFPrincipal.FormClose(Sender: TObject; var Action: TCloseAction);
begin
   if Application.MessageBox('Deseja mesmo sair do programa?', 'Confirmação',
MB_YESNO or MB_ICONQUESTION )=IDNO then
      Abort
   else
      Exit;
end;

procedure TFPrincipal.Sair1Click(Sender: TObject);
begin
   Close;
end;

procedure TFPrincipal.UseCase1Click(Sender: TObject);
begin
   tipoDiag := 'Use-Case';

   // pega informacoes para o diagrama
   FNovoDiag := TFNovoDiag.Create(self);
   FNovoDiag.ShowModal;
   FNovoDiag.Free;

   with DataModule1 do
   begin
      if (tDiag.FieldByName('tipoDiag').AsString = tipoDiag) and criou then
      begin
         // cria a janela filha que representa o diagrama
         WindowMenu := Exibir1;
         UseCase := TFUseCase.Create(self);
         UseCase.Show;
         UseCase.Caption := UseCase.Caption + ' - ' +
tDiag.FieldByName('tituloDiag').AsString;
```

```
            UseCase.codDiag := tDiag.FieldByName('codDiag').AsInteger;
            criou := false;
        end
    end
end;

procedure TFPrincipal.tbAtorClick(Sender: TObject);
begin
    if MDIChildCount <> 0 then
    begin
        if ActiveMDIChild is TFUseCase then
        begin
            { cria imagens dinamicamente }
            ator := nil;
            ator := TAtor.Create(self);
            ator.Parent := ActiveMDIChild;  { atribui o form filho que estive a tivo }
            with ActiveMDIChild as TFUseCase do
            begin
                DataModule1.tFiguras.Locate('nomeFig', 'ator', []);
                with ator do
                begin

Picture.LoadFromFile(DataModule1.tFiguras.FieldByName('figura').AsString);
                    Transparent := true;
                    ShowHint := true;
                    AutoSize := true;
                    PopupMenu := PopUpMenu1;
                    OnMouseDown := ImagemMouseDown;
                    OnClick := ImagemClick;

                    { guarda as informações sobre a figura no BD }
                    with DataModule1.tFigDiag do
                    begin
```

```
                    Append; // adiciona uma nova figura
                    FieldByName('esquerda').AsInteger := ator.Left;
                    FieldByName('topo').AsInteger := ator.Top;
                    FieldByName('altura').AsInteger := ator.Height;
                    FieldByName('largura').AsInteger := ator.Width;
                    FieldByName('codDiag').AsInteger := codDiag;  // informa a que
diagrama a figura pertence
                    FieldByName('codFigura').AsInteger                    :=
DataModule1.tFiguras.FieldByName('codFigura').AsInteger;
                    FieldByName('codClasse').AsInteger := 1; // essa é uma classe
registrada sem nome
                    Post;
                    ator.codAtor := FieldByName('codFigDiag').AsInteger;
                end; // fim de with
            end // fim de with
        end // fim de with
    end  // fim de if
  end // fim de if
end;

procedure TFPrincipal.ImagemMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { so permite arrastar se o botao esquerdo for apertado }
  begin
    if Sender is TImage then begin
      with Sender as TImage do  { trata Sender como TImage }
      begin
        BeginDrag(False);
      end;
    end;

    if Sender is TLabel then begin
      with Sender as TLabel do  { trata Sender como TLabel }
```

```
        begin
            BeginDrag(False);
        end;
    end;
end
else
begin
    if Sender is TImage then
    begin
        { se nao for arrastar (botao direito pressionado), apenas guarda
        a  imagem (que eh o Sender) para futuras operacoes }
        imagem := TImage(Sender);
        // guarda tb o codigo da figura para uso no banco de dados
        if Sender is TAtor then
        begin
            with Sender as TAtor do
            begin
                codFiguraAux := codAtor
            end
        end
        else if Sender is TUseCase then
        begin
            with Sender as TUseCase do
            begin
                codFiguraAux := codUCase;
            end
        end
        else if (Sender is TClasses) and (not(Sender is TObjetos)) then
        begin
            with Sender as TClasses do
            begin
                codFiguraAux := codClasse;
            end
```

```
            end
            else if Sender is TObjetos then
            begin
                with Sender as TObjetos do
                begin
                    codFiguraAux := codObjeto;
                end
            end
        end
        else
        begin
            { se nao for arrastar (botao direito pressionado), apenas guarda
            o texto (que eh o Sender) para futuras operacoes }
            texto := TLabel(Sender);
            if Sender is TTextos then
            begin
                with Sender as TTextos do
                begin
                    codTextoAux := codTexto
                end
            end;
        end;
        { as duas linhas de codigo a seguir guardam o ponto exato que o mouse
        foi clicado na figura, para um melhor posicionamento ao arrastar }
        posX := X;
        posY := Y;
    end;
end;

procedure TFPrincipal.tbUCaseClick(Sender: TObject);
begin
    { cria imagens dinamicamente }
    if MDIChildCount <> 0 then  { verifica se existem janelas filhas }
```

```
begin
    if ActiveMDIChild is TFUseCase then  { verifica se a janela filha ativa é um
diagrama de use-case }
    begin
      uCase := TUseCase.Create(self);
      uCase.Parent := ActiveMDIChild;  { atribui o form filho que estive a tivo }
      with ActiveMDIChild as TFUseCase do
      begin
        DataModule1.tFiguras.Locate('nomeFig', 'usecase', []);
        with uCase do
        begin

Picture.LoadFromFile(DataModule1.tFiguras.FieldByName('figura').AsString);
          Transparent := true;
          ShowHint := true;
          AutoSize := true;
          PopupMenu := PopUpMenu1;
          OnMouseDown := ImagemMouseDown;
          OnClick := ImagemClick;

          // guarda as informações da figura no BD
          with DataModule1.tFigDiag do
          begin
            Append;  // adiciona uma nova figura
            FieldByName('esquerda').AsInteger := uCase.Left;
            FieldByName('topo').AsInteger := uCase.Top;
            FieldByName('altura').AsInteger := uCase.Height;
            FieldByName('largura').AsInteger := uCase.Width;
            FieldByName('codDiag').AsInteger := codDiag;  // informa a que
diagrama a figura pertence
            FieldByName('codFigura').AsInteger :=
DataModule1.tFiguras.FieldByName('codFigura').AsInteger;
            FieldByName('codClasse').AsInteger := 1; // essa é uma classe
registrada sem nome
            Post;
```

```
                    uCase.codUCase := FieldByName('codFigDiag').AsInteger;
                end; // fim de with
            end // fim de with
        end // fim de with
    end // fim de if
end // fim de if
end;

procedure TFPrincipal.NovoProjeto1Click(Sender: TObject);
begin
    FNovoProj := TFNovoProj.Create(self);
    FNovoProj.ShowModal;
    FNovoProj.Free;
    if codProjeto <> 0 then
    begin
        // habilita e desabilita opcoes do menu principal
        NovoDiagrama1.Enabled := true;
        FecharProjeto1.Enabled := true;
        NovoProjeto1.Enabled := false;
        AbrirProjeto1.Enabled := false;
        DeletaProjeto1.Enabled := true;
        Salvar1.Enabled := true;

        // habilita e desabilita os botões da barra de ferramentas
        tbFecharProjeto.Enabled := true;
        tbNovoProjeto.Enabled := false;
        tbAbrirProjeto.Enabled := false;
        tbSalvarProjetó.Enabled := true;
    end
end;

procedure TFPrincipal.AbrirProjeto1Click(Sender: TObject);
var
```

```
  i: integer;
begin
  FAbrirProj := TFAbrirProj.Create(self);
  FAbrirProj.ShowModal;
  FAbrirProj.Free;
  with DataModule1 do
  begin
    if codProjeto <> 0 then
    begin
      // seleciona os diagramas pertencentes ao projeto escolhido
      quDiag.Close;
      quDiag.Params[0].AsInteger := codProjeto;
      quDiag.ExecSQL;
      quDiag.Open;
      // habilita e desabilita opcoes do menu principal
      NovoDiagrama1.Enabled := true;
      FecharProjeto1.Enabled := true;
      NovoProjeto1.Enabled := false;
      AbrirProjeto1.Enabled := false;
      Salvar1.Enabled := true;
      DeletaProjeto1.Enabled := true;
      // habilita e desabilita os botões da barra de ferramentas
      tbFecharProjeto.Enabled := true;
      tbNovoProjeto.Enabled := false;
      tbAbrirProjeto.Enabled := false;
      tbSalvarProjeto.Enabled := true;
      // abre os diagramas
      for i := 1 to quDiag.RecordCount do
      begin
        WindowMenu := Exibir1;
        if quDiag.FieldByName('tipoDiag').AsString = 'Use-Case' then
        begin
          UseCase := TFUseCase.Create(self);
```

```
                UseCase.Show;
                UseCase.codDiag := quDiag.FieldByName('codDiag').AsInteger; //
guarda o codigo do diagrama numa variavel
                UseCase.CarregaFiguras;
                UseCase.CarregaTextos;
                UseCase.Caption := UseCase.Caption + ' - ' +
quDiag.FieldByName('tituloDiag').AsString;
            end;
            if quDiag.FieldByName('tipoDiag').AsString = 'Classes' then
            begin
                Classes := TFClasses.Create(self);
                Classes.Show;
                Classes.codDiag := quDiag.FieldByName('codDiag').AsInteger; //
guarda o codigo do diagrama numa variavel
                Classes.CarregaFiguras;
                Classes.CarregaTextos;
                Classes.Caption := Classes.Caption + ' - ' +
quDiag.FieldByName('tituloDiag').AsString;
            end;
            if quDiag.FieldByName('tipoDiag').AsString = 'Objetos' then
            begin
                Objetos := TFObjetos.Create(self);
                Objetos.Show;
                Objetos.codDiag := quDiag.FieldByName('codDiag').AsInteger; //
guarda o codigo do diagrama numa variavel
                Objetos.CarregaFiguras;
                Objetos.CarregaTextos;
                Objetos.Caption := Objetos.Caption + ' - ' +
quDiag.FieldByName('tituloDiag').AsString;
            end;
            quDiag.Next;
        end; // fim de for
        quDiag.Close;
    end;
end // fim de with
```

```
end;

procedure TFPrincipal.FecharProjeto1Click(Sender: TObject);
var  i: integer;
begin
   // habilita e desabilita opcoes do menu principal
   NovoProjeto1.Enabled := true;
   AbrirProjeto1.Enabled := true;
   NovoDiagrama1.Enabled := false;
   FecharProjeto1.Enabled := false;
   Salvar1.Enabled := false;
   DeletaProjeto1.Enabled := false;

   // habilita e desabilita os botões da barra de ferramentas
   tbFecharProjeto.Enabled := false;
   tbNovoProjeto.Enabled := true;
   tbAbrirProjeto.Enabled := true;
   tbSalvarProjeto.Enabled := false;
   codProjeto := 0;  // indica que nao tem nenhum projeto aberto
   // fecha os diagramas
   for i := MDIChildCount - 1 downto 0 do
   begin
      MDIChildren[i].Free;
   end;
   Caption := 'UML';
end;

procedure TFPrincipal.FormCreate(Sender: TObject);
begin
   codProjeto := 0;
   criou := false;
end;
```

```
procedure TFPrincipal.DeletaProjeto1Click(Sender: TObject);
var  i: integer;
begin
   if Application.MessageBox('Deseja mesmo excluir esse projeto?', 'Confirmação',
MB_YESNO or MB_ICONQUESTION )=IDNO then
      Abort
   else
   begin
      with DataModule1.tProjetos do
      begin
         Open;
         if locate('codProj', codProjeto, []) then
         begin
            Delete;
            // habilita e desabilita opcoes do menu principal
            NovoProjeto1.Enabled := true;
            AbrirProjeto1.Enabled := true;
            NovoDiagrama1.Enabled := false;
            FecharProjeto1.Enabled := false;
            Salvar1.Enabled := false;
            DeletaProjeto1.Enabled := false;
            // habilita e desabilita os botões da barra de ferramentas
            tbFecharProjeto.Enabled := false;
            tbNovoProjeto.Enabled := true;
            tbAbrirProjeto.Enabled := true;
            tbSalvarProjeto.Enabled := false;
            codProjeto := 0; // indica que nao tem nenhum projeto aberto
            // fecha os diagramas
            for i := MDIChildCount - 1 downto 0 do
            begin
               MDIChildren[i].Free;
            end;
         end;
```

```
            Close;
        end
    end // fim de if-else
end;
procedure TFPrincipal.Classes2Click(Sender: TObject);
begin
    tipoDiag := 'Classes';

    // pega informacoes para o diagrama
    FNovoDiag := TFNovoDiag.Create(self);
    FNovoDiag.ShowModal;
    FNovoDiag.Free;
    with DataModule1 do
    begin
        if (tDiag.FieldByName('tipoDiag').AsString = tipoDiag) and criou then
        begin
            // cria a janela filha que representa o diagrama
            WindowMenu := Exibir1;
            Classes := TFClasses.Create(self);
            Classes.Show;
            Classes.Caption := Classes.Caption + ' - ' +
tDiag.FieldByName('tituloDiag').AsString;
            Classes.codDiag := tDiag.FieldByName('codDiag').AsInteger;
            criou := false;
        end
    end
end;

procedure TFPrincipal.tbClasseClick(Sender: TObject);
begin
    if MDIChildCount <> 0 then
    begin
        if ActiveMDIChild is TFClasses then
```

```
        begin
            { cria imagens dinamicamente }
            classe := nil;
            classe := TClasses.Create(self);
            classe.Parent := ActiveMDIChild;  { atribui o form filho que estive a tivo }
            with ActiveMDIChild as TFClasses do
            begin
                DataModule1.tFiguras.Locate('nomeFig', 'classe', []);
                with classe do
                begin

Picture.LoadFromFile(DataModule1.tFiguras.FieldByName('figura').AsString);
                    Transparent := true;
                    ShowHint := true;
                    AutoSize := true;
                    PopupMenu := PopUpMenu1;
                    OnMouseDown := ImagemMouseDown;
                    OnClick := ImagemClick;
//                      OnMouseUp := ClasseMouseUp;

                    { guarda as informações sobre a figura no BD }
                    with DataModule1.tFigDiag do
                    begin
                        Append;  // adiciona uma nova figura
                        FieldByName('esquerda').AsInteger := classe.Left;
                        FieldByName('topo').AsInteger := classe.Top;
                        FieldByName('altura').AsInteger := classe.Height;
                        FieldByName('largura').AsInteger := classe.Width;
                        FieldByName('codDiag').AsInteger := codDiag;  // informa a que
diagrama a figura pertence
                        FieldByName('codFigura').AsInteger                    :=
DataModule1.tFiguras.FieldByName('codFigura').AsInteger;
                        FieldByName('codClasse').AsInteger := 1; // essa é uma classe
registrada sem nome
```

```
                    Post;
                    classe.codClasse := FieldByName('codFigDiag').AsInteger;
                end; // fim de with
            end // fim de with

        end // fim de with

    end  // fim de if

end // fim de if
end;

procedure TFPrincipal.ToolButton1Click(Sender: TObject);
begin
    if MDIChildCount <> 0 then
    begin
        { cria textos dinamicamente }
        textos := nil;
        textos := TTextos.Create(self);
        textos.Parent := ActiveMDIChild;  { atribui o form filho que estive a tivo }
        with textos do
        begin
            Transparent := true;
            AutoSize := true;
            textos.Caption := 'NENHUM';
            OnMouseDown := ImagemMouseDown;
            PopupMenu := MenuTexto;

            { guarda as informações sobre a figura no BD }
            with ActiveMDIChild as TFChild do
            begin
                with DataModule1.tLabels do
                begin
```

```
                Append; // adiciona um novo texto
                FieldByName('esquerda').AsInteger := textos.Left;
                FieldByName('topo').AsInteger := textos.Top;
                FieldByName('texto').AsString := textos.Caption;
                FieldByName('codDiag').AsInteger := codDiag;    // informa a que
diagrama a figura pertence
                Post;
                textos.codTexto := FieldByName('codTexto').AsInteger;
            end; // fim de with
        end; // fim de with
    end // fim de with
  end // fim de if
end;

procedure TFPrincipal.Apagar1Click(Sender: TObject);
begin
  with DataModule1.tLabels do
  begin
    if Locate('codTexto', codTextoAux, []) then
    begin
      Refresh;
      Delete;
      Refresh;
      texto.Destroy;
    end
  end;
end;

procedure TFPrincipal.Editar1Click(Sender: TObject);
begin
  FEdTexto := TFEdTexto.Create(self);
  FEdTexto.ShowModal;
  FEdTexto.Free;
```

```
end;

procedure TFPrincipal.tbObjetoClick(Sender: TObject);
begin
   if MDIChildCount <> 0 then
   begin
      if ActiveMDIChild is TFObjetos then
      begin
         { cria imagens dinamicamente }
         objeto := nil;
         objeto := TObjetos.Create(self);
         objeto.Parent := ActiveMDIChild;  { atribui o form filho que estive a tivo }
         with ActiveMDIChild as TFObjetos do
         begin
            DataModule1.tFiguras.Locate('nomeFig', 'objeto', []);
            with objeto do
            begin

Picture.LoadFromFile(DataModule1.tFiguras.FieldByName('figura').AsString);
               Transparent := true;
               ShowHint := true;
               AutoSize := true;
               PopupMenu := PopUpMenu1;
               OnMouseDown := ImagemMouseDown;
               OnClick := ImagemClick;
//                OnMouseUp := ClasseMouseUp;
               { guarda as informações sobre a figura no BD }
               with DataModule1.tFigDiag do
               begin
                  Append;  // adiciona uma nova figura
                  FieldByName('esquerda').AsInteger := objeto.Left;
                  FieldByName('topo').AsInteger := objeto.Top;
                  FieldByName('altura').AsInteger := objeto.Height;
```

```
                    FieldByName('largura').AsInteger := objeto.Width;
                    FieldByName('codDiag').AsInteger := codDiag;  // informa a que
diagrama a figura pertence
                    FieldByName('codFigura').AsInteger                    :=
DataModule1.tFiguras.FieldByName('codFigura').AsInteger;
                    FieldByName('codClasse').AsInteger := 1; // essa é uma classe
registrada sem nome
                    Post;
                    objeto.codObjeto := FieldByName('codFigDiag').AsInteger;
                    objeto.codClasse := 1;
                  end; // fim de with
                end // fim de with
              end // fim de with
            end  // fim de if
          end // fim de if
end;

procedure TFPrincipal.Objetos2Click(Sender: TObject);
begin
    tipoDiag := 'Objetos';
    // pega informacoes para o diagrama
    FNovoDiag := TFNovoDiag.Create(self);
    FNovoDiag.ShowModal;
    FNovoDiag.Free;
    with DataModule1 do
    begin
        if (tDiag.FieldByName('tipoDiag').AsString = tipoDiag) and criou then
        begin
            // cria a janela filha que representa o diagrama
            WindowMenu := Exibir1;
            Objetos := TFObjetos.Create(self);
            Objetos.Show;
            Objetos.Caption      :=      Classes.Caption      +      '      -      '      +
tDiag.FieldByName('tituloDiag').AsString;
```

```
            Objetos.codDiag := tDiag.FieldByName('codDiag').AsInteger;

            criou := false; // reseta a variavel criou
        end
    end
end;
end.
```

Bibliografia

[**ATLE 93**] ATLEE, Joane M. and GANNON, John — *State-Based Model Checking of Event-Driven System Requirement* — IEEE Transactions on Software Engineering, Vol 19. No. 1, Janeiro de 1993;

[**BAIN 98**] BAINES, R. — *Across Disciplines: Risk, Design, Method, Process and Tools* — IEEE Software, Ed. Julho/Agostol, 1998, Vol. 15, Num. 4, pp. 26-29;

[**BASK 2001**] BASKERVILLE, Richard — *How Internet Software Companies Negotiate Quality* – IEEE Computer, May, 2001, 28-35 pp;

[**BERR 98**] BERRY, D. M.; e LAWRENCE, B. — *Requirements Engineering* — IEEE Software, Ed. Março/Abril, 1998, pp. 26-29;

[**BHAR 99**] BHARGAVA, H. K.; SRIDHAR, S; e HERRICK, C — *Beyond Spreadsheets: Tools for Build Decision Support Systems* — IEEE Computer Society, Vol. 32, Num. 3, Março de 1999, pp. 31-38;

[**BELL 95**] BELLINZONA, Roberto; FUGINI, Maria Grazia; and PERNICI, Barrbara — *Reusing Specification in OO Application* — IEEE Software, Março de 1995, pp. 65-75;

[**BEM 98**] BEM-SHAUL, Israel; GISH, James W.; ROBINSON, William — *Na Integrated Component Architecture* — IEEE Software, Setembro/Outubro de 1998, pp. 79-87;

[**BOOC 91**] BOOCH, G. — *Object-Oriented Design with Applications* — The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991;

[**BOOC 94**] BOOCH, =G. — *Object-Oriented Design with Applications* — The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994;

[**BOOC 95a**] BOOCH, and RUMBAUGH, J. — *Introduction to the Unified Method: Unifying the Booch & OMT Methods* — SIGPLAN Tutorial 30, OOPSLA '95, Austing, Texas, Outubro de 1995;

[**BOOC 95b**] BOOCH, Grady; and RUMBAUGH, James — *Unified Method for Object-Oriented Development - Overview Version 0.8.* — Rational Software Corporation, 1995;

[**BOWE 95**] BOWEN, Jonathan P. and HINCHEY, Michael G. — *Seven More Myths of Formal Methods* — IEEE Software, Julho de 1995, pp. 34-41;

[**CHEN 83**] CHEN, Bo-Shoe and YEH, Raymond T. — *Formal Specification and Verification of Distributed Systems* — IEEE Transactions on Software Engineering, Vol SE-9. No. 6, Novembro de 1983, pp. 710-722;

[**COAD 90**] COAD, P. and YOURDON, E. — *Object-Oriented Analysis* — 1ª Ed. Yourdon Press, Englewood Cliffs, New Jersey, 1990;

[**COAD 91**] COAD, P. and YOURDON, E. — *Object-Oriented Analysis* — 2ª Ed. Yourdon Press, Englewood Cliffs, New Jersey, 1991;

[**COAD 95**] COAD, P.; NORTH, David; and MAYFIELD, Mark — *Object Models Strategies, & Applications* — Yourdon Press, Englewood Cliffs, New Jersey, 1995;

[**COAD 95**] COAD, P.; NORTH, David; and MAYFIELD, Mark — *Object Models Strategies, & Applications* — Yourdon Press, Englewood Cliffs, New Jersey, 1995;

[**COLE 94**] COLEMAN, D.; Arnold, P; Bodoff, S.; Dollin, C.; Gilchrist, H.; Hayes, F.; Jeremaes, P. — *Object-Oriented Development: THE FUSION METHOD* — Prentice-Hall, 1994, 314 pp.;

[**COME 99**] COMEFORD, Richard — *The Path to Open-Sourch Systems* — IEEE Spectrum, Maio, 1999, Vol. 36, Num. 5, 25-31;

[**DAMM 2000**] DAMM, Christian H.; K. M. Hansen; M. Thomsen; M. Tyrsted – Tool Integration: Experiences and Issues in Using XMI and Component Technology – IEEE, 2000;

[**DAVI 94**] DAVID, René; ALLA, Hassane — *Petri Nets for Modeling of Dynamic Systems - A* Survey — Automatica Vol. 30, No. 2, 1994, pp. 175-202;

[**DAVI 97**] DAVIS, Alan M.; JORDAN, Kathleen; NAKAJIMA, Tsuyoshi — *Elements Underlying the Specification Requirements* — Anais de Engenharia de Software, Vol 3, J. C. Baltzer AG, Science Publish, 1997, pp. 63-100;

[**DERM 93**] DERNIAME, Jean Claude — *Life Cycle Process Suport In PCIS (Portable Commom Interface Set)* — Apresentação para Singapore CASE'93, Julho de 1993;

[**DIGR 98**] DIGRE, Tom — *Business Object Component Architecture* — IEEE Software, Setembro/Outubro de 1998, 60-69 pp.;

[**DIME 93**] DIMESTINE, Simon. — *Semantic Transfer Language as a Gateway for Methodology Portability* — Tese de Mestrado, Grand Valley State University, Abril de 1993;

[**ECKE 94**] ECKERT, G.; and GOLDER, p. — *Improving Object-Oriented Analysis* — Information and Software Technology, Vol. 36, No. 2, 1994;

[**EMBL 95**] EMBLEY, David W.; JACKSON, Robert B.; and WOODFIELD, Scott N. — *OO Systems Analysis: Is It or Isn't It?* — IEEE Software, Julho de 1995, pp. 19-33;

[**ERIK 98**] ERIKSSON, Hans-Erik; PENKER, Magnus — *UML Toolkit* — John Wiley & Sons Inc., Canada, 1998, 397 pp;

[**FICH 92**] FICHMAN, Robert G. and KEMERER, Chris F. — *Object-Oriented and Conventional Analisys and Design Methodologies* — IEEE Computer, Outubro de 1992;

[**FOWL 95**] FOWLER, Martin. —*A Comparison of Object-Oriented Analysis and Design Methods* — OOPSLA '95, 1995.

[**FREE 91**] FREEDMAN, Roy S. — *Testability of Software Components* — IEEE Transactions on Software Engineering, Vol. 17, No. 6, Junho de 1991, pp. 553-563;

[**FOLK 98**] FOLK, Michael J.; ZOELLICK, Bill; RICCARDI, Greg — *File Structures: Na Object-Oriented Approach wich C++* — Addison-Wesley, Menlo Park, CA, 1998. 724 pp.;

[**FULT 95**] FULTON, James A. — *Strategy for the Integration of Knowledge-Based Engineering Data* — Relatório de Trabalho Draft 1.2, Boeing Information and Support Services Research and Technology, Junho de 1995;

[**GALL 91**] GALLAGHER, Keith Brian and LYLE, James R. — *Using Program Slicing in Software Maintenance* — IEEE Transactions on Software Engineering, Vol. 17, No. 8, Agosto de 1991, pp. 751-761;

[**GAMM 95**] GAMMA, Erich; HELM, Richard; JOHNSON, Ralp, VLISSIDES, John — *Design Patterns: Elements of Reusable Object-Oriented Software* — Addison-Wesley, 1995, 395 pp.;

[**GHEZ 89**] GHEZZI, Carlo, MANDRIOLI, Dino; MORASCA, Sandro; PEZZÈ, Mauro — *A General Way to Put Time in Petri Nets* — Association for Computer Machinary, Artigo No. ACM 0-89791-305-1, 1989;

[**GORM 96**] GORMAN, Ian E. — *Lex and Yacc / Compiler-construction techniques for the everyday programmer* — Dr. Dobb's Journal, Fevereiro de 1996;

[**GRAH 94**] GRAHAM, Ian. - *Object Oriented Methods,* Addison-Wesley, 2ª Ed., Workingham, Inglaterra, 1994;

[**HAUS 94**] HAUSLER, P. A.; LINGER, H. C.; and TRAMMELL, C. J. — *Adopting Cleanroom software engineering with phased approach* — IBM SYSTEM JOURNAL, Vol. 33, No. 1, 1994;

[**HARM 98**] HARMON, Paul; WATSON, Mark – *Understanding UML — The Developer's Guide with a Web-Based Application in Java* — Morgan Kaufmann Publishers, São Francisco, CA, 1998, 367 pp.;

[**HELL 99**] HELLMAN, Reed — *A Semantic Approach Adds Meaning to the Web* — IEEE Computer, Dezembro de 1999, 13-16 pp.;

[**HSIA 94**] HSIA, Pei; SAMUEL, Jayarajan; GAO, Jerry; KUNG, David; TOYOSHIMA, Yasufumi; and CHEN, Chris — *Formal Approach to Scenario Analysis* — IEEE Software Mach 1994, pp. 33-41;

[**IEEE 92**] IEEE — *Trial-Use Standard Reference Model for Computing System Tool Interconnections* — IEEE Std 1175, Pub. Agosto de 1992, 151 pp.;

[**IIVA 95**] IIVARI, Juhani. — *Object-orientation as structural, functional and behavioral modelling: a comparison of six methods for object-oriented analysis* — Information and Software Technology, Vol. 37, No. 3, 1995;

[**INTR 2000**] INTRONA, Lucas; NISSENBAUM, Helen — *Definig the Web: The Politics of Search Enginhes* — Computer, Vol. 33, N. 1, Janeiro de 2000, 54-62 pp.;

[**JACK 95**] JACKSON, Michael — *Software Requirements & Specifications (a lexicon of practice, principles and prejudices)* — Addison-Wesley, 1995;

[**JACK 96**] JACKSON, Daniel — *Riquirements Need Form, Maybe Formality* — IEEE Software, Março de 1996, 21-22 pp.;

[**JACO 92**] JACOBSON, Ivar; CHRISTERSON, Magnos; JONSSON, Patrik; OVERGAARD, Gunnar — *Object-Oriented Software Engineering: A Use Case Drivem Approach* — Addison-Wesley, 1992, 528 pp.;

[**JACO 99**] JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James — *The Unified Process* — Addison-Wesley, California, Janeiro de 1999, 463 pp.;

[**JACO 99a**] JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James — *The Unified Process* — IEEE Software, Maio/Junho de 1999, 96-102 pp.;

[**JAHA 94**] JAHANIAN, Farnam and MOK, Aloysius K. — *Modechart: A Specification Language for Real-time Systems* — IEEE Transactions on Software Engineering, Vol 20 No. 12, Dezembro de 1994, pp. 933-947;

[**JASP 94**] JASPER, Robert; BRENNAN, Mike; WILLIAMSON; and CURRIER, Bill — *Test Data Generation and Feasible Path Analysis* — ACM ISSTA 94, Seattle, Washington, 1994, pp. 95-106;

[**JORG 94**] JORJESEN, Paul C.; Ericson, Carl. — *Object-Oriented Integration Testing* — Communication of the ACM, Vol 7, N. 9, Setembro de 1994;

[**JORG 95**] JORGENSEN, Paul C. — *SOFTWARE TESTING-A Craftsman's Approach* — Boca Raton, FL, CRC Press, 1995;

[**JORG 98**] JORGENSEN, P.; MEDEIROS, Álvaro F. C. — *Consolidating Control Models in IEEE Standard 1175* — ISAS'98, Orlando 12-16 Julho, 1998, Vol 1, 318-325 pp.;

[**KAIN 99**] KAINDL, Hermann — *Difficulties in the Transition from OO Analysis to Design* — IEEE Software, Outubro de 1999, 94-102 pp.;

[**KANI 2000**] KANITZ, Stephen — Revolucione a Sala de Aula — Revista Veja, 18, Outubro, 2000, pp. 23;

[**KANI 98**] KANITZ, Stephen — Iniciativa x Acabativa — Revista Veja, 11, Novembro, 2000, pp. 22;

[**KERN 78**] KERNIGHAN, Brian; RITCHIE, Denis — *The C Programming Language* — Prentice HallI, 1978;

[**KOZA 98**] KOZACZYNSKI, Wojtek; BOOCH, G. — *Component-Based Software Engineering* — IEEE Software, Setembro/Outubro de 1998, 34-36 pp.;

[**KUCE 94**] KUCERA, Henry and ROSS, Michael — *Survey of Object-Oriented Diagrams (Comparison of the Object-Oriented Graphical Notation)* — ISO/IEC JTC1 SC21 SQL/MM SOU 013, artigo para o Grupo WG11 na Reunião de Maio de 1994;

[**LAWR 96**] LAWRENCE, Brian — *Do You Really Need a Formal Riquirements?* — IEEE Software, Março de 1996, 20-22 pp.;

[**LERN 2000**] LERNER, M. Reuven. — *More About Searching* — Linux Journal, Fevereiro de 2000, vol 70, 50-54 pp.;

[**LAM 94**] LAM, Simon S. and SHANKAR, Udaya — *A Theory of Interfaces and Modules I - Composition Theorem* — IEEE Transactions on Software Engineering, Vol 20. No. 1, January 1994;

[**LING 94**] LINGER, Richard C. — *Cleanroom Process Model* — IEEE Software Mach 1994, pp. 50-58;

[**LOUN 95**] LOUNAMA, Pertti — *The Future Belongs to the Adaptable Too* —, IEEE Software Mach 1995, pp. 9;

[**LUCK 95**] LUCKHAM, David C. and VERA, James — *An Event-Based Architeture Definition Language* — IEEE Transactions on Software Engineering, Vol 21. No. 9, Setembro de 1995, pp. 717-734;

[**LUBA 93**] LUBAR, Mitchell; POTTS, Colin; and RICHER, Charles — *A Survey of Influential Object-Oriented Requirements Analysis Methods, Relatório da MCC* — 1994, 176 pp.;

[**MACH 96**] MACHADO, Felipe N. R; ABREU, Maurício — *Projeto de Banco de Dados: Uma Visão Prática* — 2ª Ed., Érica, São Paulo, 1996, 298 pp.;

[MALE 91] MALER, O.; MANNA, Z.; POWER, J. — *Real Time: Thery in Practice* — Springer, Berlin, Alemanha, 1991;

[MANS 90] Manual da Ferramenta CASE System Achitect;

[MANO 90] Manual da Ferramenta CASE Object Maker;

[McCA 2000] McCANNEL, Steve — *The Best Influences on Software Engineering* — IEEE Software, Vol. 17, N. 1, Janeiro/Fevereiro de 2000, 10-17 pp.;

[MART 95] MARTINS, Luiz M. F. ; MOURA, J. Antão B. ; MEDEIROS, Álvaro F. C. — *R-Cycle: Um Molde para o Processo de Produção, Disponibilização e Evolução de Software* — SBES'95 - IX Simpósio Brazileiro de Engenharia de Software, Outubro de 1995, Recife, Brazil;

[MEDE 94] MEDEIROS, Álvaro; SAUVE, Jacques; MOURA, Antao; NICOLETTI, Pedro. — *Aumentando a Produtividade e Qualidade em Sistemas Abertos* — Makron Books, 1994, 374 pp.;

[MEDE 96] MEDEIROS, Álvaro F. C.; JORGENSEN P.; MOURA, J. Antão; e MEDEIROS, Ismênia M. S. — *New STL Constructs to Compare Object-Oriented Development Software Specification Methodologies* — ISAS'96, Orlando 22-26 de julho, 1996;

[MEDE 98] MEDEIROS, Álvaro F. C.; JORGENSEN P.; e MOURA, J. Antao — *Object-Oriented Development Software Specification Methodologies Equivalence View Using STL Concepts* — ISAS'98, Orlando 12-16 Julho, 1998, Vol 5E, 76-82 pp.;

[MEDE 98a] MEDEIROS, Álvaro F. C. — *Relatório Técnico sobre Análise e Projetos de Sistemas II* — I Workshop do LabES, João Pessoa, CCEN-DI, 8-9 Junho, 1998, 9 pp.;

[MEDE 99] MEDEIROS, Álvaro F. C. — *Relatório Técnico sobre Projetos apresentados sobre a Disciplina de Análise e Projetos de Sistemas II* — II workshop do LabES, João Pessoa, CCEN-DI, 18-19 de agosto de 1999;

[MEDE 2000] MEDEIROS, Álvaro F. C.; MOURA, J. Antão; e JORGENSEN P. — *STL Extension to Represent Object-Oriented Software Specification Methods and to Allow Communication of Models Among CASE Tools more Fully* — 4ht Multiconference on Systemics, Cybernetics and Informatics - SCI, Orlando, Flórida, 23-26 de de Julho, 2000, Vol XI, 152-160 pp.;

[MEDE 2000a] MEDEIROS, Álvaro F. C. — *Relatório Técnico: Uso de Ferramentas Unix para Implementar Linguagem de Transferência de Semântica* — III workshop do LabES, João Pessoa, CCEN-DI, 25 de outubro de 2000, 10 pp.;

[MEDE 2000b] MEDEIROS, Álvaro F. C. — *Relatório Técnico: Projetos Apresentados sobre a Disciplina de Análise e Projetos de Sistemas II* — III workshop do LabES, João Pessoa, CCEN-DI, 25 de outubro de 2000, 8 pp.;

[MEDE 2001] MEDEIROS, Álvaro F. C. — *Artigo "Facilitando a Interoperabilidade de Ferramentas Case a partir de uma biblioteca STL" em Elaboração a ser submetido a SBES 2001;*

[MEDE 2001a] MEDEIROS, Álvaro F. C. — *Artigo "Planejando Ambientes CASE Heterogêneos a partir de Hierarquia de Conceitos em STL" em Elaboração a ser submetido a SCI 2001;*

[MEDE 2001b] MEDEIROS, Álvaro F. C. — *Relatório Técnico sobre a Implementação de EspCase* — I mostra de Trabalhos do Centro de Ciências Exatas e da Natureza em 31/01/2001;

[MEYE 2001] **MEYER,** Bertrand – *Software Engineering in the Academy* – IEEE Computer, May, 2001, 28-35 pp.;

[MOOR 99] MOORE, W. James — *An Integrated Collection of Software Engineering Standards* — IEEE Software, Novembro/Dezembro de 1999, 51-65 pp.;

[**MOUR 98**] MOURA, J. Antão; MEDEIROS, Álvaro F. C.; e BARROS, Marcelo A. — *R-Cycle - A Practical Approach for Managing Processes in the Real Life Cycle of Software Products* — ISAS'98, Orlando 12-16 Julho, 1998, Vol 1, 356-363 pp.;

[**MRTI 95**] MARTIN, James and ODELL, James J. — *Object-Oriented Methods AFOUNDATION* — Prentice-Hall, Englewood Cliffts, New Jersey, 1995;

[**NAUR 60**] NAUR, P. — *Revisited Report on the Algorithmic Language Algol 60* — Communications of the ACM, Vol. 6, Nº 1, Jan. 1960, pp. 1-17;

[**NISS 96**] NISSEN, Hans W; JEUSFELD, Manfred A.; JARKE, Matthias — *Managing Multiple Requirementss Perspectives with Metamodels* — IEEE Software, Março de 1996, 37-47 pp.;

[**OBRI 2000**] O'BRIEN, Mark John — *Creating data exchange standards with XML : a waste?* — 1st International Conference on Web Information Systems Engineering, 2000;

[**PINH 96**] PINHEIRO, Francisco A. C.; GOGOUEN, Joseph A. — *An Object-Oriented Tool for Tracing Requeriments* — IEEE Software, Março de 1996, 52-64.;

[**POUN 96**] POUNTAIN, Dick — *Best CASE Scenario: Choosing OO* Methods — Revista Byte, Agosto de 1996, 11-16 pp.;

[**PRESS 97**] PRESSMAN, Roger S. — *Software Engineering: A Practioner's Approach* — 4ª Edição, São Francisco, 1997, 852 pp.;

[**ROSS 95**] ROSSIE, Jonathan G. Jr. and FRIEDMAN, Daniel P. — *An Algebraic Semantics of Subobjects* — Anais da OOPSLA '95, Outubro de 1995, pp. 187-199;

[**RUMB 91**] RUMBAUGH, J.; BLAMA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, W. — *Object-Oriented Modeling and Design* — Prentice Hall, Englewood Cliffs, New Jersey, 1991;

[**SAND 95**] SANDER, Laurence — *Data Modeling* — Boyd & Fraser Publishing Company, Boston, 1995, 148 pp.;

[**SCHI 96**] SCHNEIDEWIND, N. F. — *Do Standards Impruve Quality?* — IEEE Software, Vol. 13, Num. 1, Janeiro, 1996, 22-24 pp.;

[**SHAN 99**] SHANKEL, Jason. — *Little Languages with Lex, Yacc, and MFC* — Dr. Dobb's Journal, Janeiro de 1999;

[**SHLA 88**] SHLAER, S. and MELLOR, S. — *Object-Oriented System Analysis: Modeling the World in Data* — Yourdon Press, Englewood Cliffs, New Jersey, 1988;

[**SHLA 91**] SHLAER, S. and MELLOR, S. — *Object Life Cycle* — Yourdon Press, Englewood Cliffs, New Jersey, 1991;

[**SHAR 95**] SHARON, David and BELL, Rodney — *TOOLS THAT BIND: Creating Integrated Environments* — IEEE Software Mach 1995, pp. 76-85;

[**SIM 2000**] SIM, Susan Elliott; KOSCHKE, Rainer — Workshop on Standard Exchange Format — ICSE, Irlanda, 2000;

[**SONG 97**] SONG, Xiping — Systematic Integration of Design Methods — IEEE Software, Vol. 14, Num. 2, 1997, 107-117 pp;

[**SOMM 97**] SOMMERVILE, Ian. — *Software Engineering* — 5th ed., Addison Wesley, 1997, 742 pp.;

[**SUTT95**] SUTTON, Stanley M. Jr.; HEIMBIGNER, Dennis; and OSTERWEIL, Leon J. — *APPL/A: A Language for Software Process Programming* — ACM Transations on Software Engineering and Methodology, Vol. 4, No. 3, Julho 1995, 221-286 pp.;

[**TEXE 98**] TEXEL, P. P.; WILIAMS, C. B. — *Use Cases Combined with BOOCH, OMT and UML* — Prentice-Hall, London, 1998, 465 pp.;

[**TOPP 93**] TOPPER, Andrew; OUELLETTE, Daniel; and JORGENSEN, Paul — *Structured Methods - Mergins Models, Techniques, and CASE* — McGraw Hill, 1993;

[**UML 96-2000**] Descrição (Versão 0.8 a 1.3) disponível na Internet no site www.rational.com/uml;

[**VINO 97**] VINOSKI, Steve — *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments* — IEEE Communications Magazine, Vol. 35, No. 2, February 1997;

[**YOUR 94**] YOURDON, Edward — *Object-Oriented System Design, An Integrated Approach* — Yourdon Press Computing Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1994;

[**WIRT 2001**] WIRTSCHAFTSINFORMATIK, Abteilung für; WIEN, Wirtschaftsuniversität — *An Overview of the Architecture of EIA's CASE Data Interchange Format (CDIF)* — wwwi.wu-wien.ac.at, Maio de 2001;

[**WIRF 90**] WIRFS-BROCK, R.; WILKERSON, B.; and WIENER, L. — *Designing Object-Oriented Software* — Prentice Hall, Englewood Cliffs, New Jersey, 1990;

[**ZEIG 2000**] ZEIGER, Paul; JEFFREY, Joe — *Ending the Holy War between Academia and Business* — IEEE Software, Vol. 17, N. 1, Janeiro/Fevereiro de 2000, 102-104 pp..