UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO

UMA METODOLOGIA PARA IMPLEMENTAÇÃO SEMI-AUTOMÁTICA DE PROTOCOLOS DE COMUNICAÇÃO

Chen Wen Lin

CAMPINA GRANDE

AGOSTO - 1992

CHEN WEN LIN

UMA METODOLOGIA PARA IMPLEMENTAÇÃO SEMI-AUTOMÁTICA DE PROTOCOLOS DE COMUNICAÇÃO

Dissertação apresentada ao Curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do grau de Mestre.

ÁREA DE CONCENTRAÇÃO: Ciência da Computação

WANDERLEY LOPES DE SOUZA

Orientador

CAMPINA GRANDE

AGOSTO - 1992

"UMA METODOLOGIA PARA IMPLEMENTAÇAO SEMI-AUTOMATICA DE PROTOCOLOS DE COMUNICAÇAO"

CHEN WEN LIN

DISSERTAÇAO APROVADA EM 28.08.1992

WANDERLEY LOPES DE SOUZA, Dr.
Orientador

JOBERTO SERGIO BARBOSA MARTINS, Dr.
Componente da Banca

MANOEL DE JESUS MENDES, Dr.
Componente da Banca

CAMPINA GRANDE, PB
AGOSTO/1992

AGRADECIMENTOS

Gostaria de agradecer, em primeiro lugar, ao meu orientador, Prof. Wanderley Lopes de Souza, pelo apoio indispensável à realização deste trabalho e pela grande paciência.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro.

Agradeço ao Departamento de Engenharia de Computação e Automação Industrial (DCA) da Faculdade de Engenharia Elétrica (FEE) da Universidade Estadual de Campinas (UNICAMP) por ter permitido a utilização de seu laboratório e de seus equipamentos para a conclusão deste trabalho.

Agradeço a todos os colegas e professores do Departamento de Sistemas e Computação, assim como todos os amigos, que colabolaram, direta ou indiretamente, para que este trabalho pudesse ser realizado. Agradeço, em particular, aos colegas André, Alfredo e Rossana pela ajuda dada no período em que permaneci em Campinas.

Finalmente, quero agradecer aos meus pais, Chen Shen Laing e Chen Yu Jing Neu, ao meu irmão, Chen Hsuen Wen, e ao meu tio, Yu K.J., pelo apoio e incentivo.

A **Salete**

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CCITT - "Comité Consultatif International Télégraphique et Téléphonique"

CEP - "Connection Endpoint"

ECMA - "European Computers Manufacturers Association"

ESPRIT - "European Strategic Programme for Research and Development in Information Technology"

ESTELLE - "Extended State Transition Language"

EWS - "Estelle Workstation"

ISO - "International Organization for Standardization"

MEFE - Máquina de Estados Finita Estendida

MEF - Máquina de Estados Finita

NBS - "National Bureau of Standards"

NC - Conexão de Rede

OSI - "Open Systems Interconnection"

RM-OSI - "Reference Model for Open Systems Interconnection"

SAA - "Systems Application Architecture"

SAP - "Service Access Point"

SEDOS - "Software Environment for the Design of Open Distributed Systems"

TC - Conexão de Transporte

TDF - Técnica de Descrição Formal

TP2 - Protocolo de Transporte Classe 2

UMA METODOLOGIA PARA IMPLEMENTAÇÃO SEMI-AUTOMÁTICA DE PROTOCOLOS DE COMUNICAÇÃO

RESUMO

Este trabalho apresenta uma metodologia para a derivação semi-automática de implementações de protocolos de comunicação especificados na Técnica de Descrição Formal (TDF) "Extended State Transition Language (Estelle)".

As principais etapas dessa metodologia são: especificação formal do protocolo, validação da especificação, geração do código de implementação e complementação do código de implementação. A implementação, gerada na linguagem de programação C, é obtida utilizando-se o conjunto integrado de ferramentas "Estelle Workstation (EWS)".

Essa metodologia é aplicada ao Protocolo de Transporte Classe 2 do modelo de referência "Open Systems Interconnection (OSI)".

A METHODOLOGY FOR SEMIAUTOMATIC IMPLEMENTATION OF COMMUNICATION PROTOCOLS

ABSTRACT

This work presents a methodology for the derivation of semi-automatic implementations of communication protocols specified in the Formal Description Technique (FDT) Extended State Transition Language (Estelle).

The main steps of this methodology are: formal specification of the protocol, validation of the specification, generation of the execution code and completion of the code. The output code, generated in the C programming language, is obtained by the use of EWS (Estelle Workstation) integrated tools.

This methodology is applied to the Transport Protocol Class 2 of the Open Systems Interconnection (OSI) reference model.

1. Introdução

A diminuição dos custos dos computadores possibilitou a sua utilização crescente, distribuindo os ambientes informatizados. Em conseqüência, surgiu a necessidade de se interligar esses ambientes dispersos geograficamente através de uma rede de computadores.

Cada fabricante implementou inicialmente a sua própria solução para conectar seus computadores. Entretanto, a conexão entre sistemas heterogêneos era extremamente difícil. Esse problema só poderia ser resolvido se houvesse uma padronização dos procedimentos de comunicação.

Denomina-se protocolo de comunicação ao conjunto de regras que garante uma troca ordenada das mensagens entre entidades comunicantes.

No final da década de 70, órgãos internacionais de padronização, tais como o "Comité Consultatif International Télégraphique et Téléphonique (CCITT)", a "International Organization for Standardization (ISO)", o "National Bureau of Standards (NBS)" e a "European Computers Manufacturers Association (ECMA)", desenvolveram esforços para padronizar os protocolos de comunicação. Em 1983, o "Reference Model for Open Systems Interconnection (RM-OSI)" foi aprovado oficialmente pela ISO, através do padrão internacional ISO 7498 [ISO 83a], sendo mais tarde acatado pelo CCITT, através da recomendação X.200 [CCIT 84].

Para um melhor entendimento do modelo OSI, a seguinte terminologia é ilustrada na Figura 1.1:

(a) serviço(N): são facilidades que a camada(N) oferece às entidades (N+1);

(b) pontos de acesso aos serviços ("Service Access Points - SAP(N)"): são interfaces lógicas entre as camadas adjacentes onde os serviços(N) são oferecidos às entidades(N+1);

(c) protocolo(N): é o conjunto de regras e formatos, que determinam o comportamento das entidades(N), quando estas comunicam-se, sincronizam-se e operam de forma concorrente através dos SAPs(N);

(d) conexão(N): as informações entre entidades(N+1) são trocadas numa conexão(N), que é estabelecida na camada(N) usando um protocolo(N);

(e) ponto terminal de conexão("Connection End Point - CEP(N)"): é o terminal de uma conexão(N).

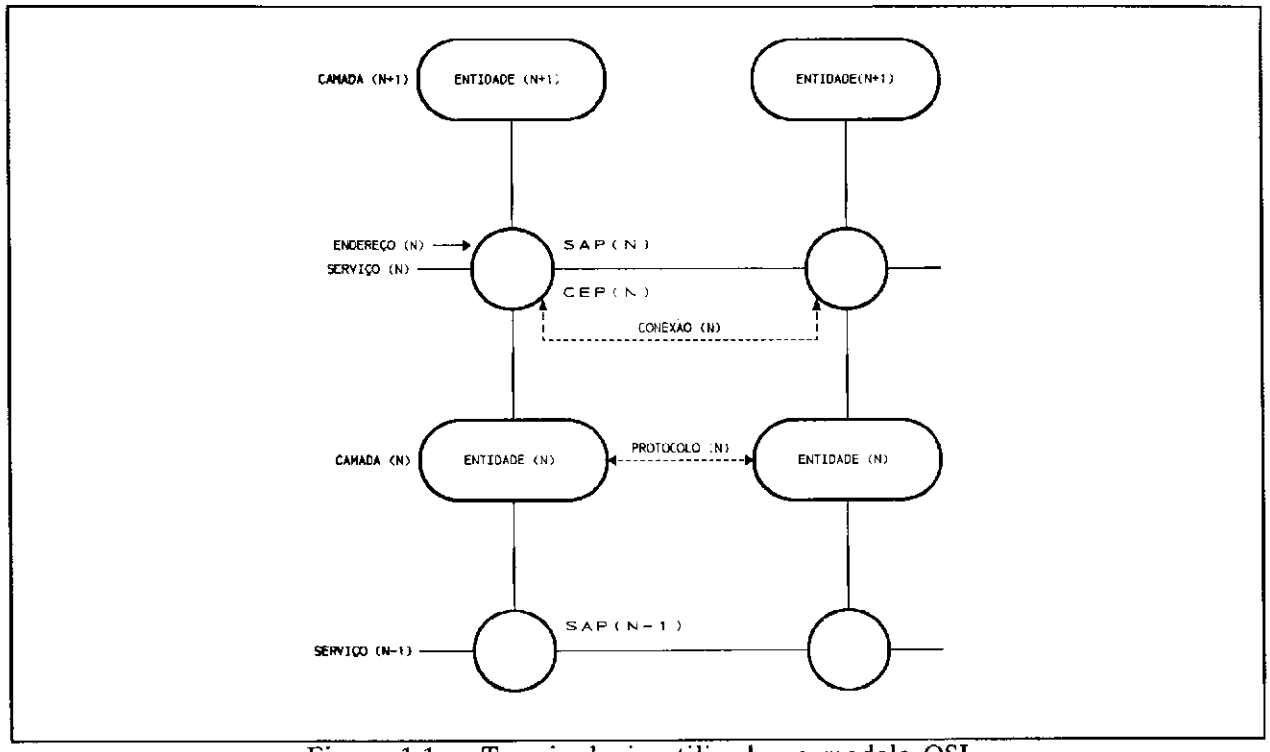

Figura 1.1 Terminologia utilizada no modelo OSI

A arquitetura do modelo OSI (Figura 1.2) é dividida em sete camadas, cada uma delas com um conjunto de funções bem definidas. Cada camada(N) utiliza os serviços providos pela camada(N-1), para fornecer um serviço de melhor qualidade à camada(N+1).

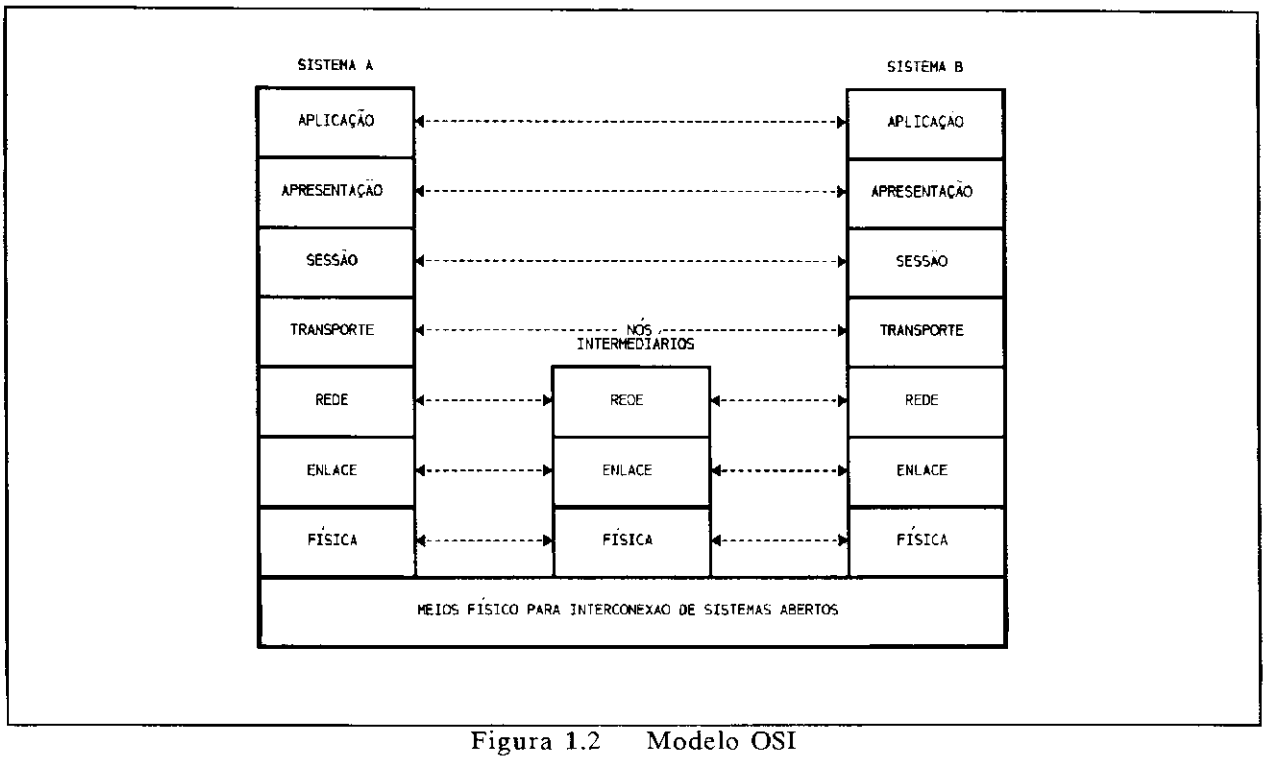

Figura 1.2 Modelo OSI

As sete camadas do modelo OSI podem ser resumidas da seguinte forma :

(a) FÍSICA: define as características mecânicas, elétricas, funcionais e os procedimentos para ativar, manter e desativar as conexões físicas para a transmissão de bits entre as entidades da camada de enlace;

(b) ENLACE: contém as funções para detectar, e, possivelmente, corrigir os erros provenientes da camada física. Permite também fornecer à camada de rede a capacidade de controlar o chaveamento de circuitos da camada física;

(c) REDE: fornece os meios necessários para a troca de unidade de dados entre entidades pares de transporte, através de conexões de rede, estabelecendo um circuito virtual ou utilizando serviços de datagrama. Executa também as funções de roteamento e controle de congestionamento da subrede;

(d) TRANSPORTE: fornece uma transferência transparente de dados (fim-a-fim e confiável) para as entidades de sessão, procurando corrigir os erros provenientes da camada de rede e otimizando a relação custo/benefício;

(e) SESSÃO: permite organizar e sincronizar o diálogo e gerenciar a troca de dados entre entidades da camada de apresentação;

(f) APRESENTAÇÃO: tem por objetivo resolver as diferenças sintáticas entre os sistemas abertos comunicantes, compatibilizando as representações das estruturas de dados;

(g) APLICAÇÃO: fornece serviços diretamente às aplicações dos usuários. Faz também o gerenciamento dessas aplicações e o gerenciamento das atividades de comunicação do sistema.

Os protocolos das camadas FÍSICA, ENLACE e REDE constituem os protocolos de BAIXO NÍVEL. Os protocolos das demais camadas constituem os protocolos de ALTO NÍVEL.

A natureza distribuída e heterogênea, dos sistemas computacionais ligados em rede, exige a existência de protocolos bem estruturados e bem especificados. O projeto e implementação de protocolos de comunicação, além dos problemas inerentes a todo projeto de desenvolvimento de software, apresenta algumas particularidades [Bock 83]:

(a) necessidade de garantir a compatibilidade entre seus diversos componentes;

(b) implantação desses componentes por grupos distintos de técnicos em organizações diferentes;

(c) dificuldade de compreensão do funcionamento do sistema como um todo, devido ao paralelismo das operações executados nos seus diferentes componentes.

Os primeiros protocolos (por exemplo, o protocolo BSC da IBM) foram especificados informalmente, utilizando-se linguagens naturais. A experiência tem mostrado que esse tipo de especificação gera ambigüidades, permitindo diferentes interpretações para a mesma descrição de um objeto.

A utilização de uma Técnica de Descrição Formal (TDF) para a especificação de serviços e de protocolos de comunicação, além de gerar especificações não ambíguas, facilita posteriormente as tarefas de verificação, implementação e teste desses protocolos.

A ISO criou o grupo de trabalho ISO TC97/SC21/WG1/Ad hoc group on FDT, posteriormente dividido em três subgrupos, para o desenvolvimento de TDFs. O subgrupo B ficou responsável pelo desenvolvimento de uma TDF baseada no conceito de Máquina de Estados Finita Estendida (MEFE), dando origem à técnica "Extended State Transition Language (Estelle)". A TDF Estelle tornou-se um padrão internacional pela ISO em 1989.

Neste trabalho, é proposta uma metodologia para a obtenção semi-automática de protocolos a partir de especificações formais, realizadas com a TDF Estelle. Essa metodologia foi aplicada ao Protocolo de Transporte Classe 2 (TP2). A partir de uma especificação formal do TP2 em Estelle, foi obtida uma implementação, utilizando-se o conjunto de ferramentas integradas "Estelle Workstation (EWS)". A utilização de ferramentas, para a obtenção de implementações a partir de especificações formais, permite gerar um código impessoal, transportável e conforme à especificação.

No capítulo 2, é ilustrado o ciclo de desenvolvimento dos protocolos. Algumas técnicas de especificação formal são apresentadas, com seus métodos de verificação e validação pertinentes. São abordadas também algumas metodologias de implementação de protocolos.

O capítulo 3 descreve a TDF Estelle e o capítulo 4 descreve a ferramenta EWS. O protocolo de transporte classe 2 é descrito no capítulo 5.

O capítulo 6 descreve a metodologia proposta, sendo que o processo de obtenção da especificação e o processo de obtenção do código da implementação são ilustrados utilizando-se o TP2. O código gerado, a partir da especificação, e o código das rotinas de suporte são analisados. Um estudo sobre a adaptação desse código a qualquer ambiente operacional é também apresentado.

O capítulo 7 encerra esta dissertação, apresentado os resultados obtidos e as principais conclusões.

2. Desenvolvimento de Protocolos de Comunicação

O ciclo de desenvolvimento de um protocolo de comunicação pode ser realizado através das seguintes etapas (Figura 2.1) [LoSt 88]:

(a) formulação de intenções, onde são definidas as principais características do protocolo;

(b) especificação informal, escrita em linguagem natural e contendo uma descrição detalhada do software a ser produzido;

(c) especificação formal, realizada através de uma TDF, o que a torna uma referência confiável;

(d) implementação, ou seja, obtenção de um código executável a partir da especificação formal.

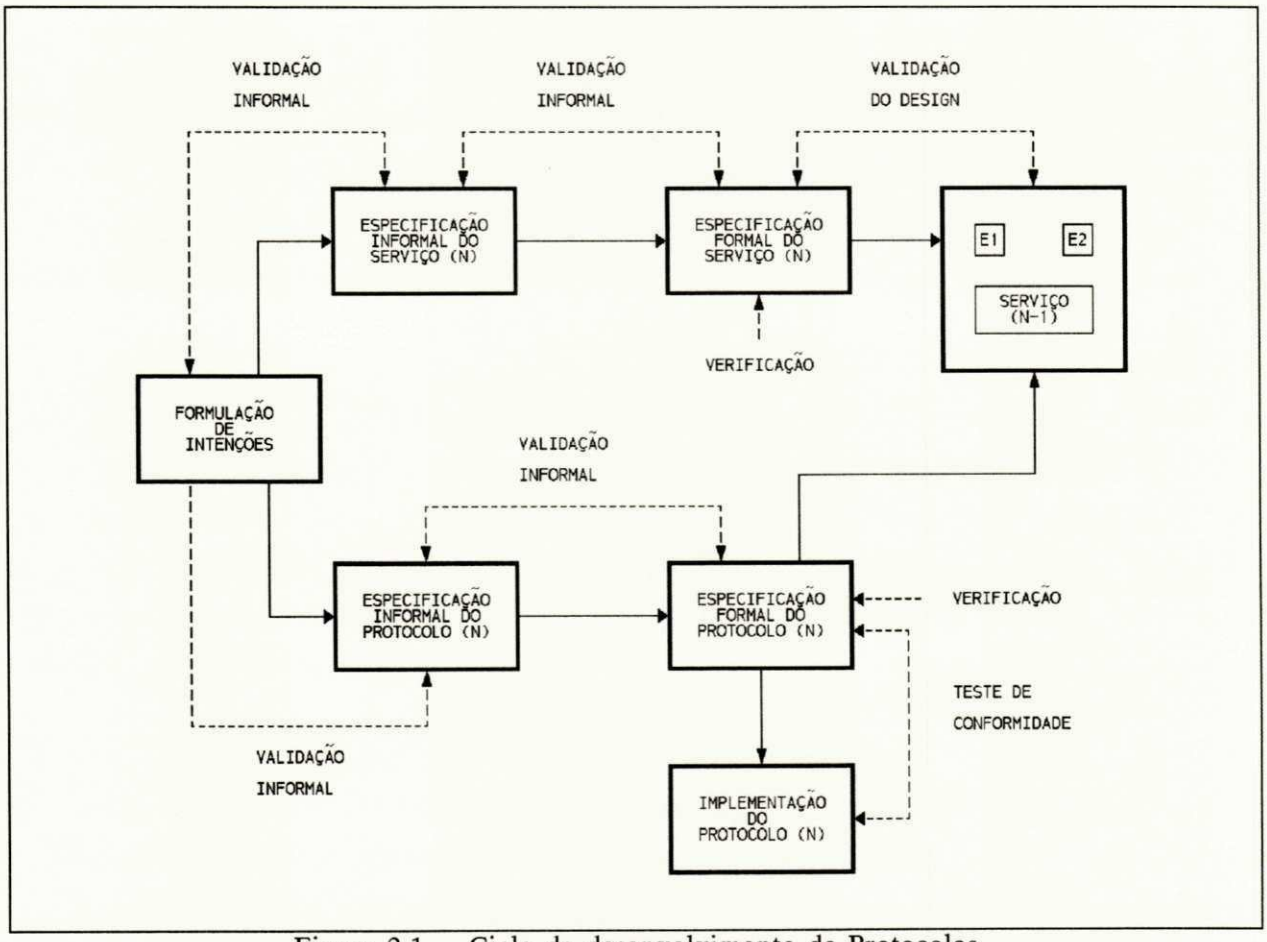

Figura 2.1 Ciclo de desenvolvimento de Protocolos

Associadas a estas etapas, podemos identificar algumas atividades de validação:

(a) validação informal, onde devem ser verificadas informalmente a conformidade das intenções do especificador em relação às especificações geradas;

(b) verificação das propriedades globais e específicas, utilizando-se métodos baseados em raciocínio lógico;

(c) validação do design do protocolo, para verificar se o funcionamento conjunto das entidades satisfaz à especificação do serviço;

(d) teste de conformidade da implementação em relação à especificação formal.

A validação tem por objetivo garantir a consistência das especificações e implementações. Ao investigar o comportamento dos protocolos, as técnicas de validação procuram assegurar-lhes uma maior confiabilidade.

2.1. Especificação Formal de Protocolos

A especificação de um software é muito importante no seu processo de desenvolvimento. Estudos realizados por Basili e Perricone [Bape 84] demonstraram que grande parte dos erros dos programas são causados por falhas na especificação (erros da própria especificação ou erros de interpretação). Dessa forma, surge a necessidade da utilização de técnicas que permitam gerar especificações precisas e corretas.

A utilização de uma TDF para a especificação de um protocolo elimina as ambigüidades de uma especificação informal, permitindo a derivação de implementações compatíveis.

Além disso, facilita as demais etapas do ciclo de desenvolvimento. A possibilidade de se verificar formalmente as principais propriedades da especificação diferencia os métodos formais dos informais e semi-formais. A especificação pode também ser utilizada para geração automática da implementação e geração automática dos cenários de teste.

O processo de obtenção de uma especificação formal pode ser baseado em refinamentos sucessivos, partindo-se da especificação informal. A especificação formal das camadas do modelo OSI envolve as especificações do serviço e do protocolo.

A especificação do serviço de uma camada (N) é baseada na descrição, por um observador externo, de uma caixa preta sujeita às trocas de primitivas de serviço com a camada superior. Na especificação do serviço são definidas as primitivas, a ordem de execução dessas primitivas e os efeitos da execução de cada primitiva.

Na especificação do protocolo(N) é descrito o comportamento das entidades que se comunicam, sincronizam-se e operam de forma concorrente através de um SAP(N).

2.1.1. Técnicas de Descrição Formal

A fim de fornecer descrições claras e concisas das camadas do modelo OSI, as TDFs devem dispor de algumas características especiais:

(a) alto grau de abstração, o que permite uma independência da especificação em relação aos métodos de implementação;

(b) poder de expressão, o que permite a descrição das características dos serviços e protocolos das sete camadas do modelo OSI de forma hierárquica e modular;

(c) poder de análise, o que permite a verificação formal das propriedades desejadas.

As principais TDFs propostas, para a especificação de protocolos de comunicação, podem ser agrupadas em três grandes categorias:

(a) técnicas baseadas em modelos de transição;

(b) técnicas baseadas em linguagens de programação;

(c) técnicas híbridas.

As TDFs baseadas em modelos de transição são eficazes para descrever aspectos de controle dos protocolos. As técnicas mais conhecidas são as baseadas em Máquina de Estados Finita (MEF) e Redes de Petri.

Uma **rede de Petri** é uma quádrupla <**L,T,E,S**>, onde **L** é um conjunto finito de nós, **T** é um conjunto de transições, **E** é a função de entrada e **S** é a função de saída. Uma ficha é atribuída a um nó desde que a condição, associada a esse nó, esteja satisfeita. A ocorrência de uma transição implica na remoção de uma ficha de cada um dos nós de entrada e na alocação de uma ficha a cada um dos nós de saída. O estado da rede, num determinado instante, é dado pela distribuição das fichas em seus nós. Segundo [DANT 80], as Redes de Petri são mais indicadas no estágio inicial da especificação dos protocolos, porque elas estão mais próximas de uma linguagem natural.

Uma **MEF** com saída é definida por uma quíntupla (**X,E,S,T,A**), onde **X** é um conjunto finito de estados, **E** é um conjunto finito de entradas, **S** é um conjunto finito de saídas, **T** é a função de transição de estado (**T: ExX -> X**) e **A** é a função de saída (**A: ExX -> S**).

Um protocolo pode ser facilmente descrito através de uma MEF com saída, uma vez que:

(a) a uma entidade pode ser atribuído um conjunto finito de estados;

(b) a partir de uma entrada (ou recebimento de interação da camada adjacente) o protocolo realiza algum processamento e, em conseqüência, gera uma saída (ou envio de interação para a camada adjacente). Uma mudança de estado também pode estar associada ao resultado desse processamento.

As linguagens de programação de alto nível podem também ser utilizadas para especificar serviços e protocolos. Esse tipo de TDF é eficaz para a descrição dos aspectos relativos às estruturas de dados dos protocolos. Contudo, as especificações produzidas são normalmente muito detalhadas, o que restringe as alternativas de implementação.

Técnicas baseadas em linguagens de programação que empregam notações mais abstratas também têm sido utilizadas para a especificação de protocolos. Exemplos desse tipo de técnica são lógica temporal e tipos abstratos de dados.

As técnicas híbridas combinam as vantagens das duas categorias apresentadas. A parte de controle dos protocolos pode ser especificada empregando-se um modelo de transição e extensões (variáveis de contexto e procedimentos), descritas numa linguagem de programação de alto nível, podem ser adicionadas.

Uma técnica híbrida típica utiliza uma MEF com saída com poucos estados (estados de controle) e um conjunto de variáveis auxiliares (estados adicionais). A cada transição é associada uma condição, que pode ser composta por um conjunto de cláusulas habilitadoras, e uma ação, que pode ser composta por um conjunto

de rotinas de processamento. Esse tipo de MEF é denominado Máquina de Estados Finita Estendida (**MEFE**).

2.2. Validação de Protocolos

Validação é qualquer atividade que visa aumentar o grau de confiabilidade de uma especificação, design ou implementação. Isto é importante pois, durante o desenvolvimento de um sistema, à medida que a especificação vai sendo detalhada, erros podem ser introduzidos.

As técnicas de validação compreendem: verificação, simulação e teste.

2.2.1. Verificação de Protocolos

Durante a geração da especificação formal de um protocolo, erros podem ser introduzidos na especificação. A especificação também pode não cobrir todas as situações de operação possíveis. Dessa forma, é necessário verificar a especificação para livrá-la desses problemas.

A verificação de um protocolo dá-se através da análise das interações entre as entidades da camada. Como exemplo de propriedades a serem verificadas, podemos citar: ausência de impasse, completitude, realização de progresso, terminação, correção parcial e estabilidade.

As Técnicas de Verificação Formal (TVFs) para as TDFs baseadas em modelos de transição utilizam métodos de **exploração de estados**. O grande problema desses métodos é a possibilidade de explosão de estados, inviabilizando qualquer tipo de análise. Como exemplo, podemos citar a Análise de Alcançabilidade.

As TVFs para as TDFs baseadas em linguagem de programação utilizam técnicas baseadas em **prova de programa** (Prova por Asserções, Lógica Temporal e Execução Simbólica).

A **Prova por Asserções** consiste em introduzir asserções em lugares apropriados da especificação, antes ou depois de determinados comandos. As asserções são usadas para descrever valores das variáveis. Para simplificar a verificação, deve-se procurar decompor a especificação em partes.

A verificação da propriedade progresso, através da prova por asserções, é difícil. Para facilitar essa verificação, a prova por asserções foi estendida com a introdução de operadores temporais.

A verificação da especificação por **Lógica Temporal** segue as mesmas linhas da técnica de Prova por Asserções. O emprego dessas técnicas dificulta a automatização da prova de programas, já que exigem uma boa dose de criatividade do projetista.

Execução simbólica tem sido utilizada na automação da verificação de especificações. O objetivo dessa técnica é construir uma Árvore de Prova. Um nó da árvore representa uma classe de estados do protocolo num dado instante. A raiz representa a classe de estados iniciais e as folhas representam os estados terminais. Os valores das variáveis de interesse são armazenados num vetor. A verificação das propriedades num determinado nó é realizada analisando-se os componentes desse vetor.

As TVFs para as TDFs híbridas utilizam também métodos híbridos. Por exemplo, técnicas de exploração de estados podem ser aplicadas ao modelo de transição e prova de programa pode se aplicada às variáveis de contexto e procedimentos.

2.2.2. Simulação e Teste

A simulação consiste em exercitar uma entidade (especificação, design ou implementação), objetivando a detecção de erros.

Os testes normalmente são utilizados para verificar a conformidade entre a especificação de um protocolo e a sua implementação.

Simulação e teste são técnicas que não proporcionam resultados definitivos, pois a investigação do protocolo é limitada a alguns cenários definidos a priori. Entretanto, tais técnicas permitem detectar uma série de erros em protocolo complexos, outorgando um certo grau de confiabilidade à entidade analisada.

2.3. Validação do Design

O objetivo da validação do design é verificar se o protocolo está atendendo às necessidades que levaram ao seu desenvolvimento. Tais necessidades estão expressas na especificação informal do serviço.

A validação do design (Figura 2.2) pode ser realizada simulando-se as especificações formais do serviço e do protocolo e submetendo-as a um conjunto de cenários. A detecção de erros é efetuada com a comparação dos traços obtidos durante a simulação do protocolo e do serviço.

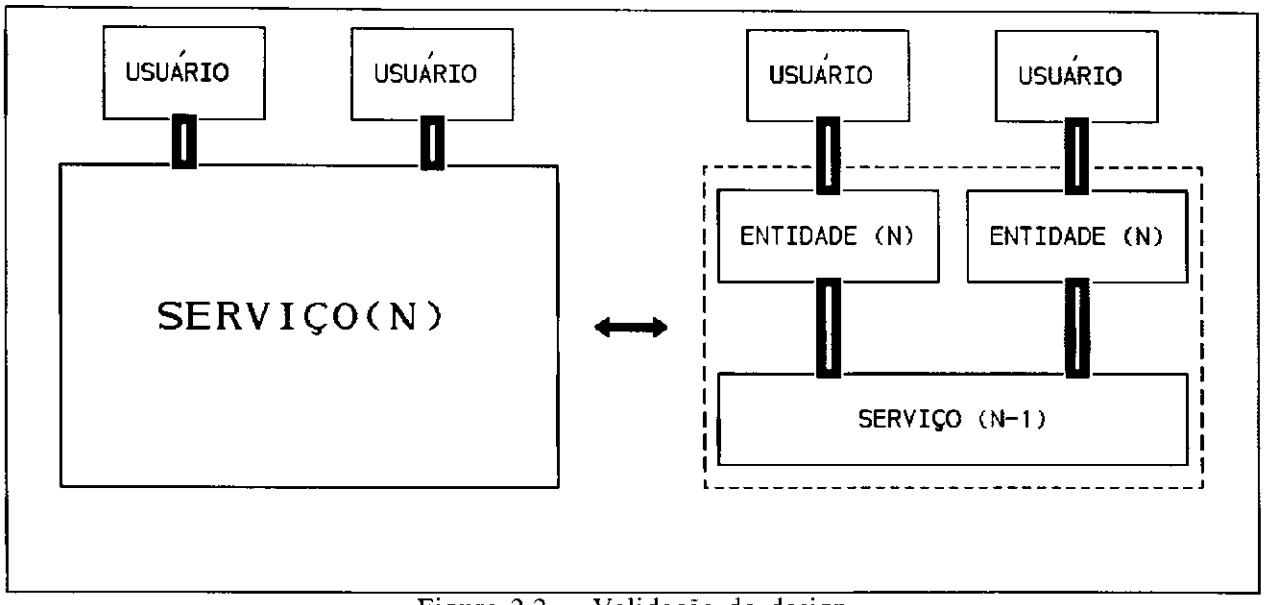

Figura 2.2 Validação do design

2.4. Implementação de Protocolos

Concluída a etapa de validação da especificação e do design do protocolo, essa especificação torna-se uma referência confiável para a derivação de uma implementação.

À medida que a especificação é refinada, ela se aproxima cada vez mais da implementação. Nesta fase, diversas decisões de implementação são tomadas: valores de certos parâmetros (temporização, tamanho da unidade de dados do protocolo, etc), estratégia de retransmissão, estratégia de reconhecimento, estratégia de atribuição de créditos, estratégia para localização do usuário, estratégia para negociação da classe de serviços e de facilidades, formatos de parâmetros (endereço, etc.), estratégias para tratamento de erros, etc.

A implementação pode ser incorporada ao ambiente operacional das seguintes formas:

(a) como um processo do sistema operacional;

(b) no núcleo do sistema operacional, obrigando o implementador a ser um especialista desse sistema;

(c) num processador front-end, isolando a implementação, mas a um custo elevado.

Um modelo geral de implementação (Figura 2.3) é sugerido em [NBS 81]:

(a) cada entidade da especificação deve ser mapeada para um conjunto de rotinas, correspondentes às transições, e um conjunto de procedimentos e primitivas;

(b) as informações sobre as entidades são armazenadas na Estrutura de Controle da Entidade;

(c) o Processador de Interface identifica a chegada de eventos provenientes das camadas adjacentes e envia interações às camadas adjacentes;

(d) as Rotinas de Eventos da Interface lêem e constroem as estruturas de dados (Estrutura de Dados de Eventos de Interface) com informações sobre o evento, além de acionar o escalonador de funções de transição;

(e) O Escalonador de Funções de Transição examina o estado da entidade na Estrutura de Controle da Entidade e consulta a Tabela de Transições para obter o conjunto de transições disparáveis. O Escalonador ativa a rotina correspondente à transição disparável.

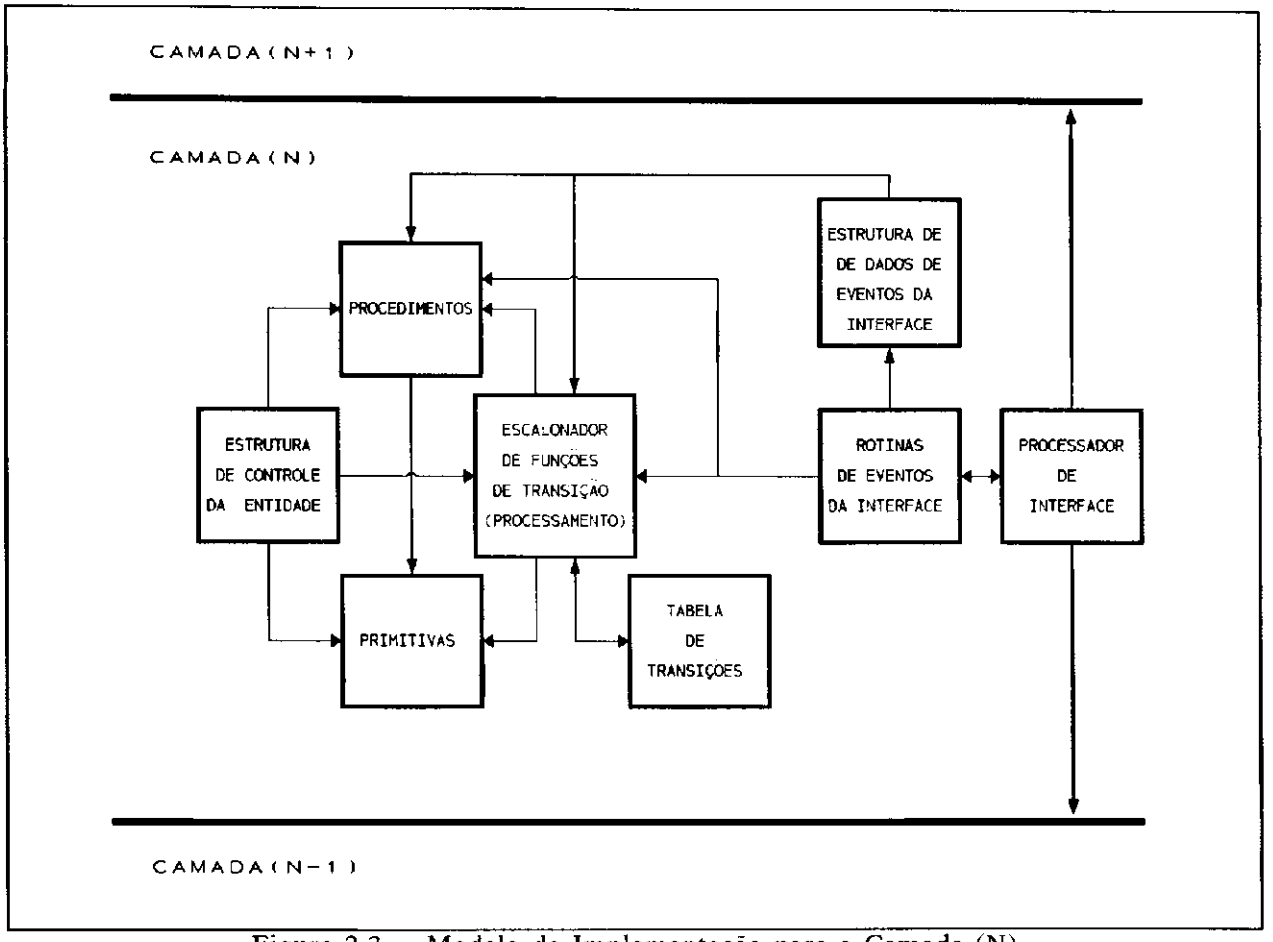

Figura 2.3 Modelo de Implementação para a Camada (N)

Em geral, os protocolos de alto nível são implementados em software e os de baixo nível em hardware. Neste trabalho, interessa-nos apenas a implementação de protocolos de alto nível.

A implementação de um protocolo deve ser o mais transportável possível, devido à complexidade do software e a seu custo de desenvolvimento.

Em [GoMo 91], é proposta uma arquitetura de protocolos OSI de alto nível para as plataformas da Systems Application Architecture (SAA) da IBM. Os códigos correspondente às camadas de Transporte, Sessão e Apresentação são transportáveis. Uma interface entre as camadas do modelo OSI e o sistema operacional, denominada Base, fornece as facilidades requisitadas pelas camadas: comunicação entre processos, serviços de entrada e

saída, gerência de buffer, etc. Essa interface é específica para cada plataforma da SAA.

2.4.1. Implementação Semi-Automática de Protocolos

Sistemas que produzem automaticamente um código numa linguagem de programação, a partir de uma especificação, são denominados de geradores automáticos de código ou compiladores de especificações.

Exemplos de geradores automáticos de códigos são os geradores de "parser" dos compiladores (por exemplo, YACC), as ferramentas CASE e os compiladores de especificações formais.

Nos compiladores de especificações formais, somente as rotinas correspondentes à especificação são geradas automaticamente. Algumas rotinas auxiliares devem ser implementadas manualmente. Por isso, denomina-se de implementação semi-automática à geração automática de apenas parte do código da implementação.

Uma implementação de protocolo, obtida com o auxílio de um compilador de especificações formais, deve possuir as seguintes partes:

(a) código correspondente à especificação;

(b) rotinas de interface com o sistema operacional;

(c) rotinas de suporte.

As rotinas de suporte são genéricas e operam como uma biblioteca de funções. Tais rotinas servem para escalonar a execução dos componentes da especificação, para gerenciar buffer e para dar suporte ao código gerado. As rotinas de suporte são

geralmente fornecidas pelo fabricante do compilador de especificações.

As rotinas de interface devem ser codificadas de acordo com o ambiente onde o protocolo irá operar. Tais rotinas permitem o acesso, do código gerado automaticamente, às facilidades do sistema operacional, garantindo a transportabilidade desse código.

A geração automática do código, a partir da especificação, facilita o teste de conformidade da implementação em relação à especificação, pois erros humanos não podem ser introduzidos durante a geração do código da implementação.

Dessa forma, uma implementação, obtida automaticamente a partir de uma especificação validada, contém um código altamente confiável, impessoal e transportável.

2.5. Testes de Implementação

A implementação de um protocolo pode apresentar erros, principalmente quando esse código foi gerado manualmente. Para dar maior confiabilidade à implementação, uma série de testes deve ser realizada. Consideramos aqui apenas os testes dos protocolos de alto nível.

Os testes procuram verificar a conformidade entre a implementação e sua especificação. Um conjunto de cenários de teste é definido a priori. A implementação é exercitada por um conjunto de entradas e as saídas são comparadas aos resultados esperados.

Idealmente, os testes da implementação devem ser realizados utilizando-se a rede onde a implementação atuará. Quando isso não

é possível, a implementação é testada simulando-se as condições de operação da rede.

Nos testes realizados com a arquitetura da Figura 2.4, o Interpretador de cenários, utilizando um arquivo de cenários, simula o usuário da camada. Um Gerador de Erros e Exceções é inserido entre a camada de rede e a camada de transporte para produzir situações anômalas.

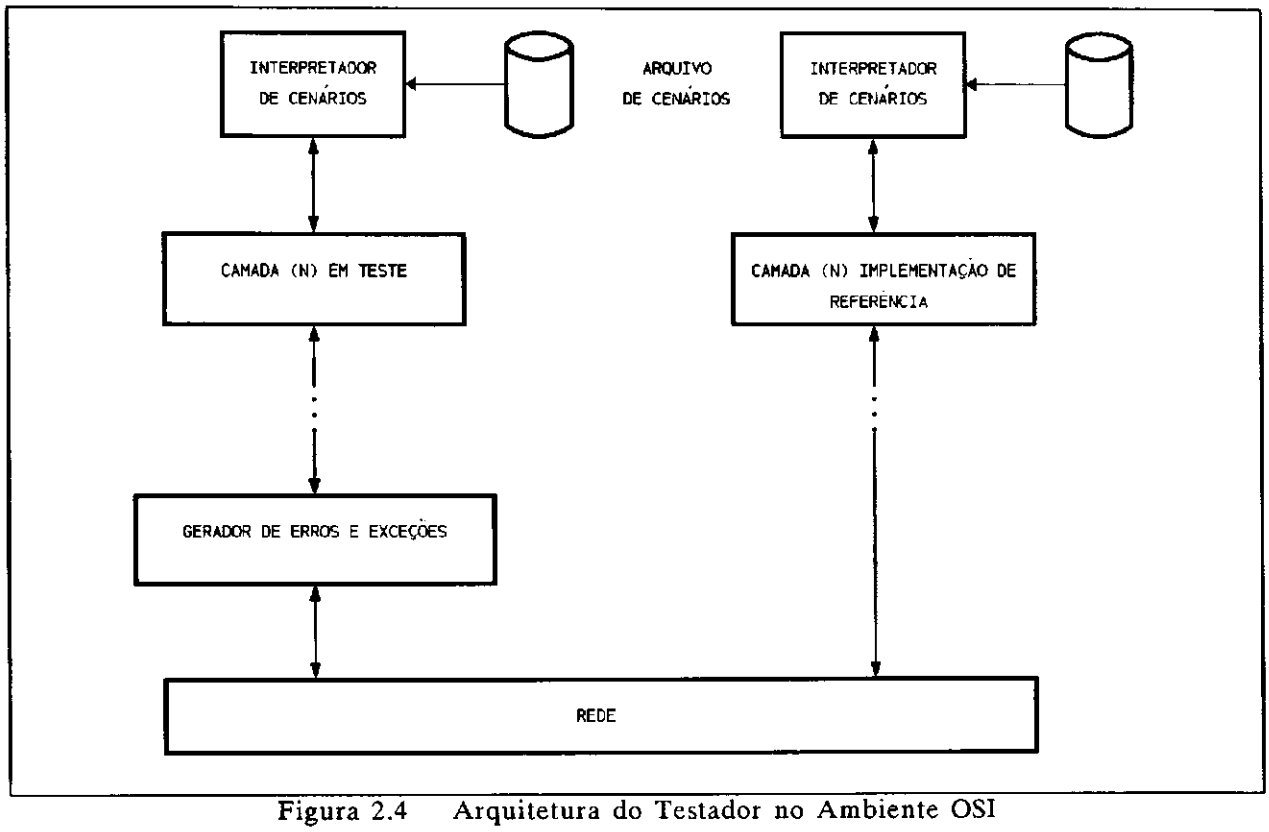

Figura 2.4 Arquitetura do Testador no Ambiente OSI

Nos testes realizados com a arquitetura da Figura 2.5, o testador envolve completamente a camada em teste, simulando as camadas superiores e inferiores.

As geração manual de seqüências de teste, além de fatigante, pode produzir cenários incompletos, já que algumas seqüências importantes podem ser simplesmente esquecidas.

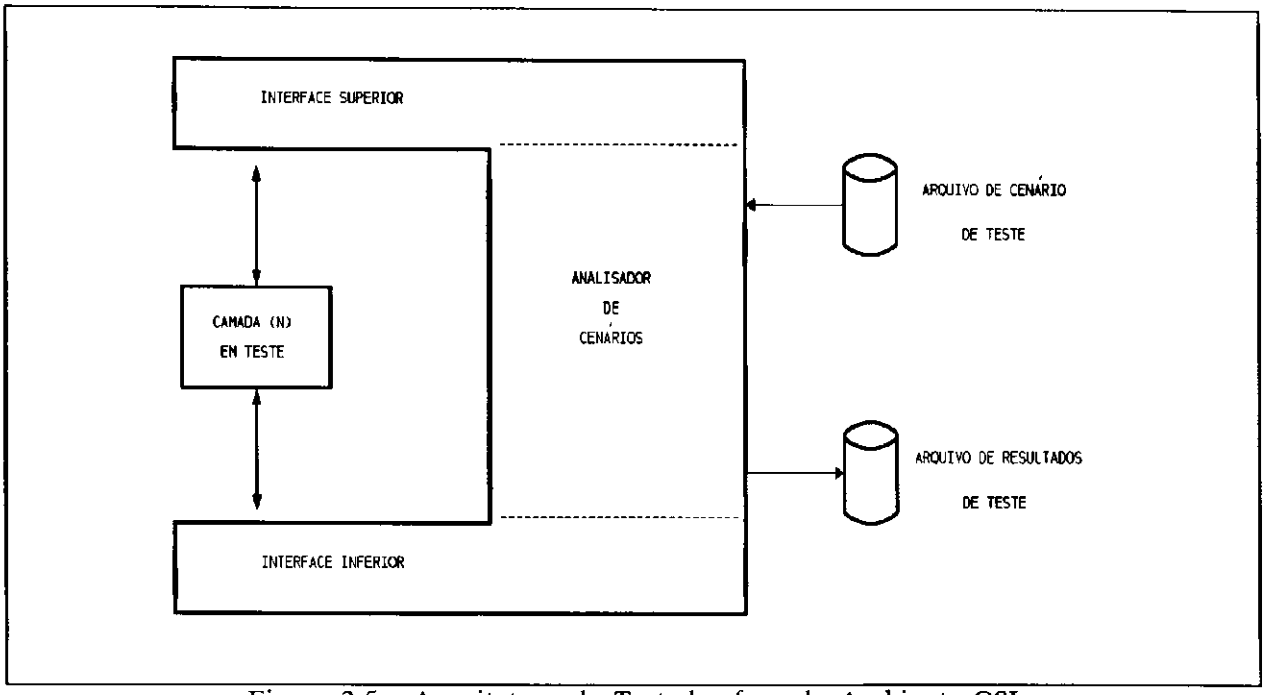

Figura 2.5 Arquitetura do Testador fora do Ambiente OSI

A geração automática de seqüências de teste, baseadas em especificações de protocolos através de MEFs, é proposta em [SaBo 82], [LiMa 83] e [ARCG 84]. Durante a exploração dos estados, é possível determinar que saídas são produzidas a partir das entradas aplicadas. A maior dificuldade na geração automática reside na seleção das seqüências de teste significativas.

Nos testes realizados com a rede, um dos problemas é a sincronização entre os dois interpretadores de cenários, principalmente se a seqüência de teste for gerada automaticamente. Isso se deve à possível ocorrência de falhas nas camadas inferiores, o que provoca uma interrupção na seqüência de transições. Por exemplo, dado duas transições consecutivas, se apenas o interpretador local tomou parte da primeira transição (pois a rede perdeu-a), o interpretador local ficará fora de sincronização com o remoto.

3. Extended State Transition Language (Estelle)

O modelo subjacente à TDF Estelle é baseado numa MEFE (Figura 3.1). Estelle possui uma sintaxe, próxima da linguagem de programação Pascal [ISO 83b], e uma semântica operacional formalmente definida.

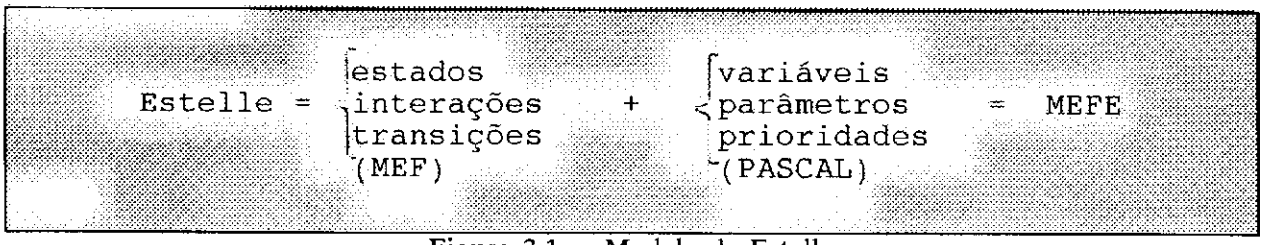

Figura 3.1 Modelo de Estelle

Desenvolvida para a especificação formal dos protocolos de comunicação, Estelle já foi utilizada também para a especificação de outros tipos de sistemas, tais como comunicação móvel, centrais telefônicas e circuitos digitais.

3.1. Arquitetura

Uma especificação em Estelle (Figura 3.2) é constituída por um conjunto de módulos hierarquicamente estruturados. Módulos podem ser refinados em submódulos, o que permite realizar a especificação de um mesmo sistema em vários níveis de abstração.

Cada módulo é representado por uma caixa preta com portas de entrada e saída (pontos de interação). Os módulos podem ser interligados, através de seus pontos de interação, via canais de comunicação bidirecionais. Os pontos de interação dos módulos filhos também podem ser vinculados aos pontos de interação dos módulos pais. Associado a cada ponto de interação, há uma fila de comprimento infinito. Isso permite que a comunicação entre os módulos seja assíncrona.

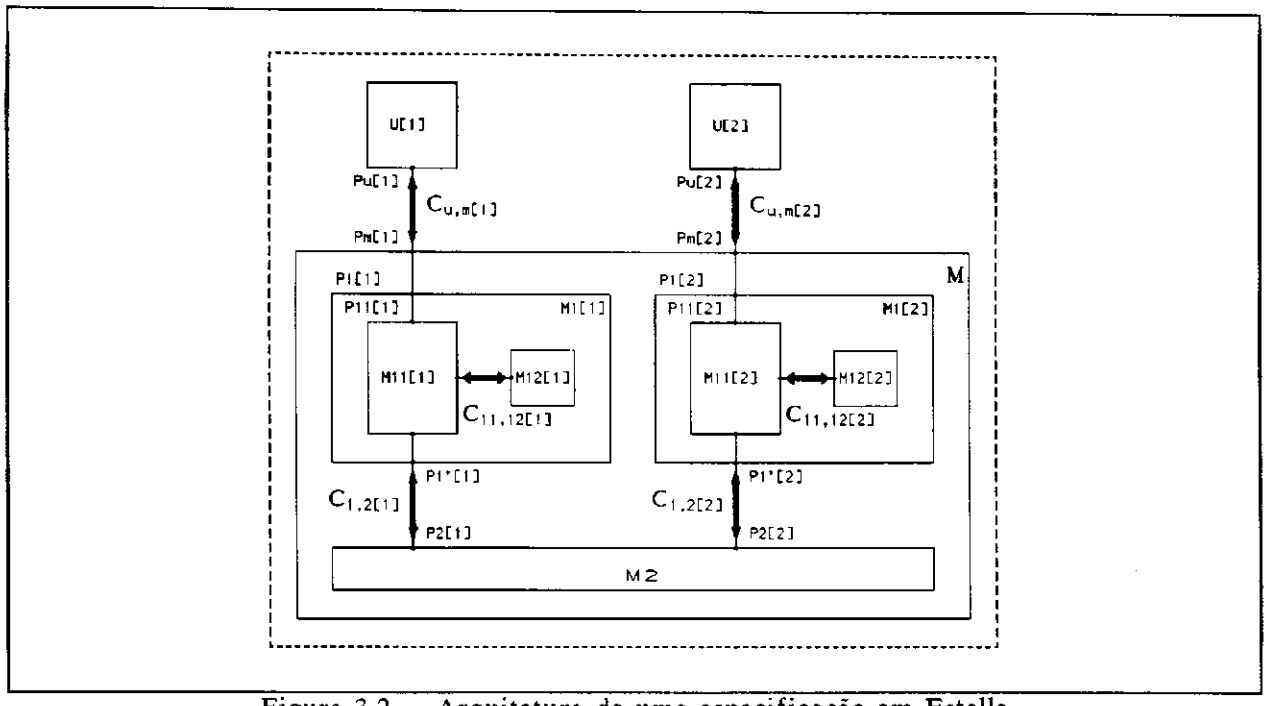

Figura 3.2 Arquitetura de uma especificação em Estelle

Um módulo pode ser ativo ou não. Ele será ativo se possuir transições. Módulos inativos são utilizados para auxiliar na estruturação da especificação. Um módulo pode ou não ter um dos seguintes atributos: **systemprocess**, **systemactivity**, **process** ou **activity**. Os princípios para a atribuição dos módulos são:

- todo módulo ativo deve possuir atributo;
- subsistemas (ou seja, módulos **systemprocess** ou **systemactivity**) podem ser aninhados somente dentro de um módulo sem atributo (inativo);
- módulos **activity** e **process** são aninhados no interior de um subsistema;
- módulos **systemprocess** e **process** podem ser refinados em módulos **process** e **activity**;
- módulos **systemactivity** e **activity** podem ser refinados somente em módulos **activity**.

Em relação ao paralelismo associado à execução dos módulos de uma especificação, os seguintes princípios são adotados:

- subsistemas são executados em paralelo assincronamente, ou seja, cada subsistema elege e mantém um conjunto de transições em execução e esses conjuntos evoluem independentemente;

- no interior de um subsistema, as transições do módulo pai têm preferência para execução;

- no interior de um módulo **systemprocess**, os módulos descendentes podem ser executados em paralelo sincronamente, ou seja, as transições são iniciadas ao mesmo tempo e todas devem terminar antes que um novo conjunto possa ser eleito para execução;

- no interior de um módulo **systemactivity**, os módulos descendentes não podem ser executados em paralelo.

3.2. Sintaxe

Uma especificação em Estelle é constituída pelo cabeçalho da especificação e pelo corpo da especificação.

No cabeçalho da especificação (Figura 3.3), é definido o nome da especificação e podem ser definidos o atributo (**systemprocess** ou **systemactivity**), a disciplina de fila padrão (**common queue** ou **individual queue**) e a unidade de tempo para a cláusula **delay**. Uma especificação que possui um atributo é constituída de um único subsistema, já que, segundo as regras de atribuição dos módulos, ela não pode ser refinada em subsistemas. Caso a especificação contenha mais de um subsistema, ela não deve ter atributo.

```
specification <ident-especificação> <atributo>;
 default   <disciplina-fila>;
 timescale <identificador>;

 <corpo-especificação>
end.
```

Figura 3.3 Sintaxe para definição de uma especificação

O corpo de uma especificação pode ser constituído por: declarações, inicializações (**initialize**) e transições (**trans**). Na parte de declarações, são definidas as constantes (**const**), os tipos de dados (**type**), os canais (**channel**), o cabeçalho (**module**) e o corpo (**body**) dos módulos, os pontos de interação internos (**ip**), as variáveis tipo módulo (**modvar**), as variáveis (**var**), as variáveis de estado (**state**), as variáveis contendo conjunto de estados (**stateset**), e as subrotinas (**function/procedure**).

A definição das constantes (**const**), dos tipos (**type**), das variáveis (**var**) e subrotinas (**function/procedure**) segue a mesma sintaxe da linguagem Pascal. As subrotinas não devem referenciar objetos em Estelle nem utilizar comandos Estelle. As outras restrições aplicadas a parte Pascal de Estelle são:

- as funções não devem ter efeitos colaterais;
- os apontadores somente podem ser utilizados na parte Pascal;
- uso restrito do comando goto;
- as funções **read** e **write**, o tipo **file** e **conformant arrays** não podem ser utilizados.

A construção **channel** de Estelle permite especificar as primitivas de comunicação independentemente da especificação dos módulos. Na especificação de um canal (Figura 3.4) são declaradas as primitivas, acompanhadas de seus parâmetros, e os papéis que os módulos, a serem conectados às extremidades desse canal,

deverão desempenhar. Assim sendo, a primitiva ident-Interação1 pode ser enviada pelo módulo que desempenhar o papel1 e deve ser recebida pelo módulo que desempenhar o papel2.

```
channel <ident-canal> (<papel1>, <pepel2>);
  by <papel1>: <Ident_Interação1> (<parâmetros>:<tipo>, ...);
               <Ident_Interação2> (<parâmetros>:<tipo>, ...);
               ...
  by <papel2>: <Ident_InteraçãoN> (<parâmetros>:<tipo>, ...);
               ...
  by <papel1>,
     <papel2>: <Ident_InteraçãoX> (<parâmetros>:<tipo>, ...);
               ...
```

Figura 3.4 Sintaxe para definição de um canal

A definição de um módulo contém duas partes: o cabeçalho e o corpo.

Se um módulo contiver um atributo ele deve ser declarado no seu cabeçalho (Figura 3.5). A especificação do cabeçalho de um módulo é baseada na descrição dos seus pontos de interação (linhas 2-5). A cada ponto de interação é agregado um canal e o papel que o ponto desempenha em relação ao canal é explicitado. O tipo de fila, que pode ser individual (**individual queue**) ou compartilhado com outros pontos de interação do mesmo módulo (**common queue**), é declarado.

Um módulo filho pode permitir que seu pai tenha acesso (ler/escrever) às suas variáveis. Essa permissão é concedida através da exportação dessas variáveis, o que também é realizado no cabeçalho do módulo (linha 7 da Figura 3.5).

```
1 module <ident-modulo> <atributo> (<parametros>:<tipo>,...);
2  ip <ident-portal>:<ident_canal> (<papel>) <disciplina-fila>;
3     ...
4     <ident-portaN>: array [<lim-inf> .. <lim-sup>] of
5                     <ident_canal> (<papel>) <disciplina-fila>;
6
7  export <ident-var>:<tipo>; ...;
8 end;
```

Figura 3.5 Definição do cabeçalho do módulo

O corpo de um módulo (Figura 3.6) pode conter três partes: declarações, inicializações e transições. A sintaxe para a definição desse corpo é a mesma que foi utilizada na definição do corpo da especificação.

```
body <ident-corpo> for <ident-modulo>;
  <declarações>
  <inicializações>
  <transições>
end;
```

Figura 3.6 Definição do corpo de um módulo

Na parte de declarações (Figura 3.7) podem ser definidos os objetos a serem manipulados, as subrotinas, os pontos de interação internos e os submódulos (caso o módulo tenha sido refinado). Constantes, tipos, variáveis e subrotinas são declarados (linhas 1, 3, 23 e 31), utilizando-se a sintaxe da linguagem Pascal. A declaração de variáveis do tipo módulo (linha 18 a 20) prepara a criação das instâncias dos módulos filhos. A variável **state** (linha 26), do tipo enumerado, é utilizada para a declaração dos estados de controle da MEF, enquanto que **stateset** (linha 29) é utilizada para a declaração de conjuntos de estado de controle.

```
01 <declaração de constantes>
02 ...
03 <declaração de tipos>
04 ...
05 <declaração de canais>
06 ...
07 <declaração de cabeçalho-módulo>
08 ...
09 <declaração corpo-módulo>
10 ...
11 (* declaração de pontos de interação internos*)
12 ip <ident-portal>:<ident_canal> (<papel>) <disciplina-fila>;
13    ...
14    <ident-portaN>: array [<lim-inf> .. <lim-sup>] of
15                    <ident_canal> (<papel>) <disciplina-fila>;
16 ...
17 (* declaração instância de módulos *)
18 modvar <ident-var-módulo>: <ident-módulo>;
19       <ident-var-módulo>: array [<lim-inf> .. <lim-sup>] of
20                           <ident-módulo>;
21 ...
22 (* declaração de variáveis *)
23 var <ident-var>: <tipo>; ...
24 ...
25 (* declaração das variáveis de estado *)
26 state <ident-estado1>, <ident-estado2>, ...;
27 ...
28 (* declaração das variáveis conjunto de estados*)
29 stateset <ident-conj-estado>=[<ident-estado1>, ...];
30 ...
31 <declaração de subrotinas>
```

Figura 3.7 Parte de declarações do corpo de um módulo

Na parte de inicialização (Figura 3.8), são atribuídos os valores iniciais ao estado de controle e às variáveis de estado adicionais (variáveis Pascal), no caso do módulo possuir transições.

```
1  initialize
2  to <estado de controle inicial>
3  begin
4     ... (* inicialização de variáveis Pascal *)
5  end;
```

Figura 3.8 Parte de inicializações do corpo de um módulo

Instâncias de módulos filhos, conexões entre pontos de interação dessas instâncias e vinculações desses pontos a pontos de interação do módulo pai podem ser criadas, estaticamente, na parte de inicializações do módulo pai (Figura 3.9).

```
1  initialize
2  to <ident-estado>
3  begin
4    init    <criação de instâncias dos filhos>
5    connect <conexão entre pontos de interação>
6    attach  <vinculação de pontos de interação>
7    ...
8  end
```

Figura 3.9 Parte de inicializações do corpo do módulo pai

A parte de transições (Figura 3.10) descreve o comportamento interno do módulo. Cada transição é composta por condição e ação. A condição é constituída de uma ou mais cláusulas habilitadoras. A ação é constituída de construções próprias a Estelle e de comandos Pascal, respeitadas as restrições descritas anteriormente.

```
01  (* condição *)
02  trans
03    from <estado vigente>
04    when <interação de entrada>
05    provided <predicado booleano>
06    priority <nº inteiro>
07    delay <min, max>
08    (* ação *)
09    to <próximo estado>
10    begin
11      output  <interação de saída>
12      init    <criação de instâncias dos filhos>
13      connect <conexão de pontos de interação>
14      attach  <vinculação de pontos de interação>
15      ... (* comandos *)
16    end;
```

Figura 3.10 Parte de transições do corpo de um módulo

Uma transição é disparável se estiver habilitada e se possuir a mais alta prioridade entre as transições habilitadas. Transições que não possuem a cláusula **when** são ditas espontânea. Na avaliação da condição de uma transição, valores por default (Figura 3.10) são atribuídas às cláusulas inexistentes.

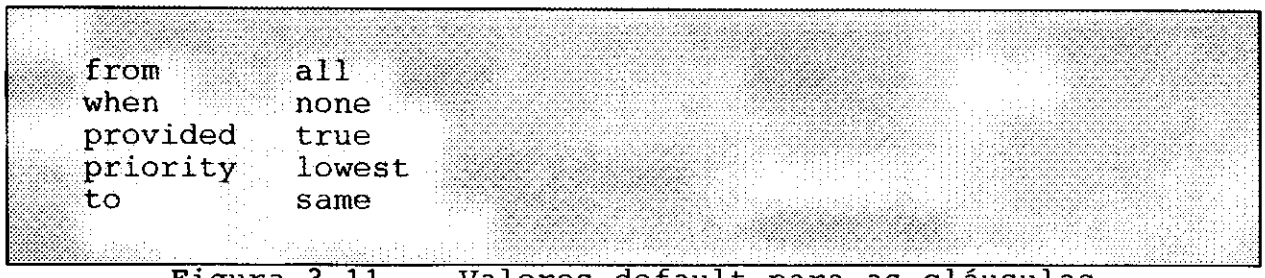

Figura 3.11 Valores default para as cláusulas

Uma transição espontânea pode ser retardada através da cláusula **delay**. Dois tempos de atraso (mínimo e máximo) podem ser associados a essa cláusula. Uma transição habilitada, que possui a mais alta prioridade e cujo tempo de atraso decorrido é superior ao mínimo e inferior ao máximo, é dita opcionalmente disparável.

A ação (linha 9 a 13) pode provocar a alteração do estado de controle e das variáveis Pascal e pode gerar interações de saída. Instâncias de módulos filhos (**init**), conexões entre pontos de interação dessas instâncias (**connect**) e vinculações desses pontos a pontos de interação do módulo pai (**attach**) podem ser criadas, dinamicamente, por ações de transições do módulo pai. Da mesma forma, outras transições do módulo pai podem executar ações respectivamente opostas (**release**, **disconnect** e **detach**).

Outras construções de Estelle podem ser empregadas nas partes de inicializações e transições (e.g., **any**, **all**, **forone**, **terminate**). Maiores detalhes sobre essas construções e sobre a própria linguagem podem ser encontrados em [ISO 88a].

4. Estelle Workstation (EWS)

Tendo por objetivo a obtenção de um conjunto integrado de ferramentas para a TDF Estelle, EWS [EWS 89] foi desenvolvido por um consórcio de empresas durante o período 1986-1989, no âmbito do "European Strategic Programme for Research and Development in Information Technology (ESPRIT)", projeto "Software Environment for the Design of Open Distributed Systems (SEDOS)/Estelle Demonstrator" (No 1265). Desenvolvido para as estações SUN e APPOLO, EWS é um ambiente composto pelas seguintes ferramentas (Figura 4.1):

- editor orientado para a sintaxe de Estelle (EWSEDIT);
- analisador de especificações (EWSTRANS);
- gerador de código C (EWSGEN)
- simulador de especificações (SIMULATOR/DEBUGGER)
- biblioteca de rotinas de suporte para o código da implementação (ESKIMO).

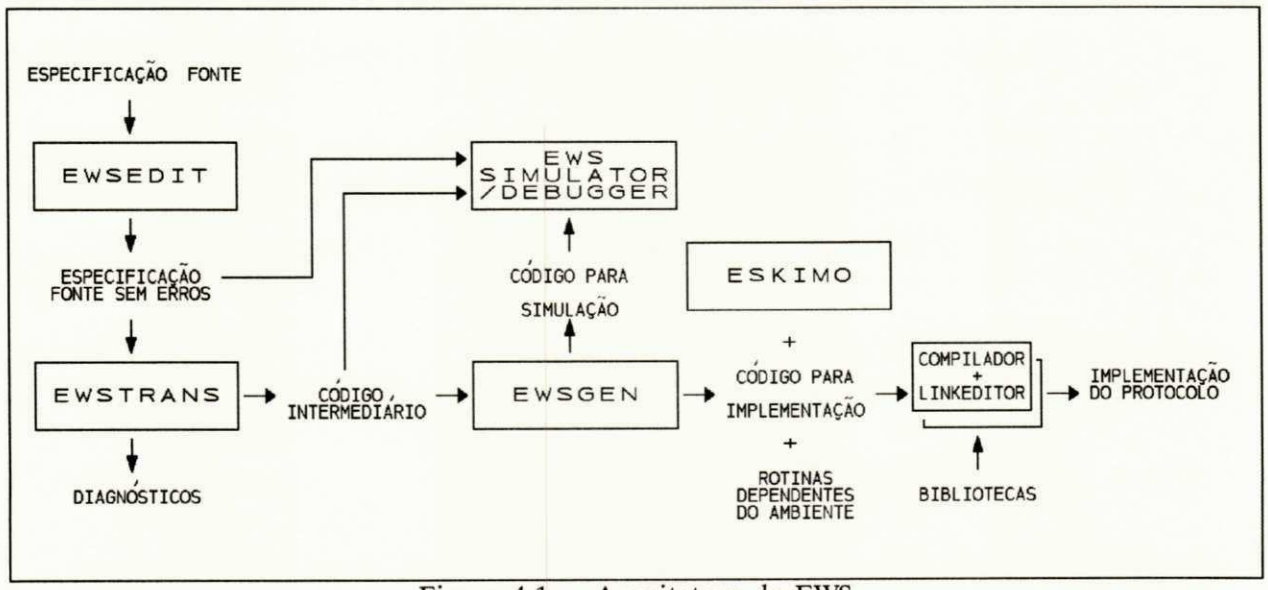

Figura 4.1 Arquitetura do EWS

4.1. EWSEDIT

EWSEDIT foi construído pela Expert Software Systems N.V., utilizando a ferramenta CASE chamada MIRA. É um editor que realiza simultaneamente a análise léxico-sintática da especificação. Os erros semânticos não são detectados. Para executar EWSEDIT, digita-se o comando **goewsedit <spec>**, onde <spec> é o nome da especificação.

Um especificador experiente em Estelle pode dispensar EWSEDIT e utilizar qualquer outro editor de texto. Entretanto, essas especificações devem ser submetidas a EWSEDIT, para ficarem no formato apropriado para o simulador.

Uma unidade de edição corresponde a uma linha completa de comandos (com "tokens" do tipo comando, variável, etc). Os comandos da linguagem podem ser digitados caractere a caractere, ou podem ser selecionados a partir de um menu sensível ao contexto.

A verificação sintática é realizada a cada unidade de edição. O texto é formatado automaticamente, inserindo-se as tabulações apropriadas.

A saída produzida por EWSEDIT é um arquivo ASCII e esse editor também lê arquivos ASCII produzidos por outros editores. Entretanto, somente especificações sintaticamente corretas podem ser gravadas no EWSEDIT.

O menu (Figura 4.2) oferece as seguintes opções: EDIT, FILE, TEXT, CLIP, VIEW e OPTS.

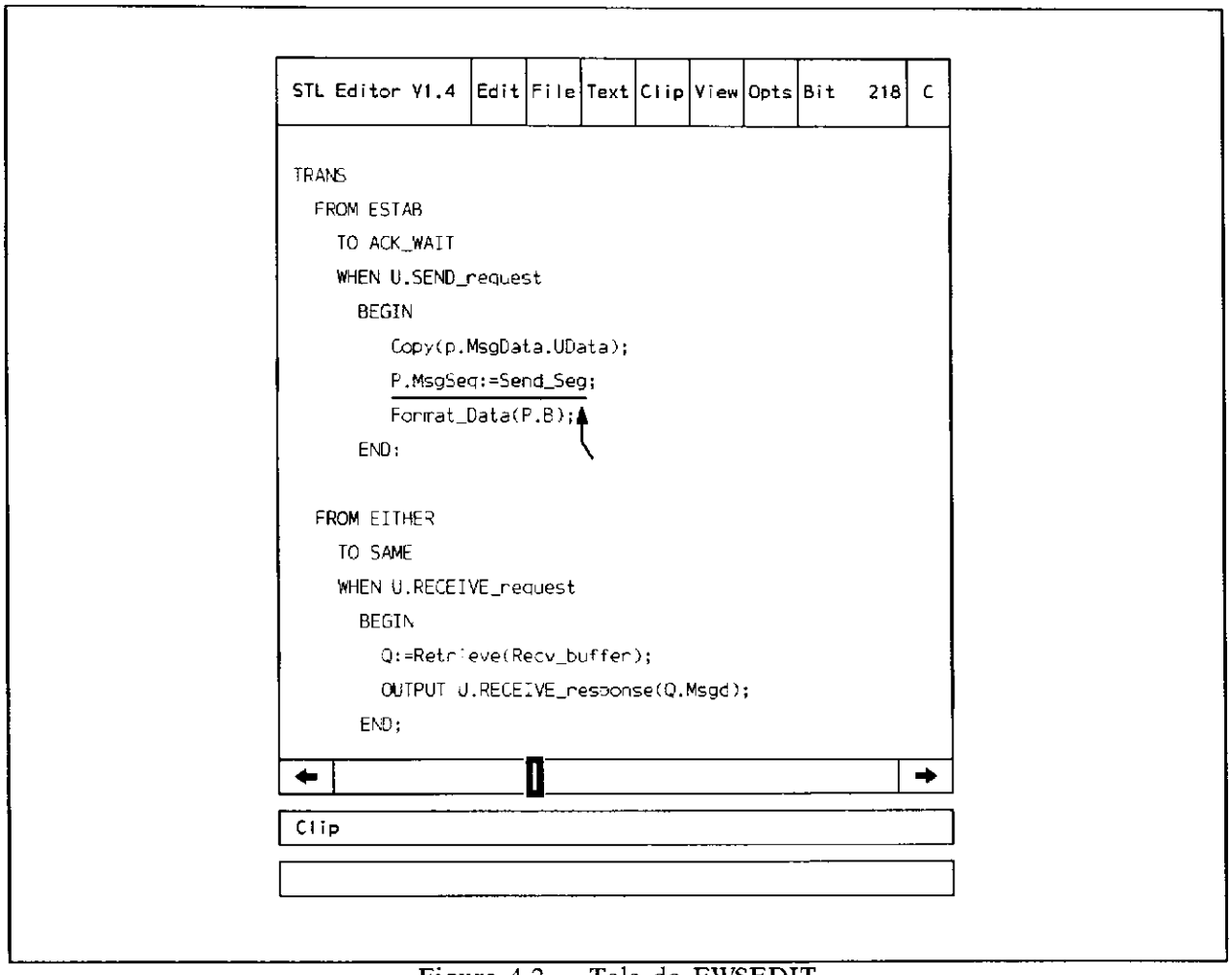

Figura 4.2 Tela do EWSEDIT

O menu **EDIT** permite percorrer a especificação (MOVE DOWN, MOVE UP, UP 1 LINE, ON TOP, GOTO e MARK), localizar (WHEREIS) e substituir (EXCHANGE) textos e abandonar o EWSEDIT (QUIT).

O menu **FILE** permite realizar as operações de leitura e gravação em disco:

- inserção de um arquivo na posição anterior à posição vigente (INSERT);
- substituição da unidade selecionada por um arquivo (REPLACE);
- inserção de um arquivo após a posição vigente (APPEND);

- gravação da unidade selecionada num arquivo (COPYn);

- gravação da especificação (SAVE).

O menu **TEXT** oferece os recursos para editar o texto da especificação. O comando INSERT permite incluir as unidades de edição. O comando ERASE permite apagar as últimas unidades de edição digitadas. Para finalizar a inserção de texto, as alterações podem ser aceitas (ACCEPT) ou rejeitadas (REJECT).

Para alterar os trechos de uma especificação, seleciona-se a unidade a ser alterada, altera-se o conteúdo da unidade (REPLACE) ou insere-se novas unidades (APPEND/INSERT) e, finalmente, confirmar-se as alterações (ACCEPT) ou não (REJECT).

O menu **CLIP** (Clipboard) permite atualizar o conteúdo da área de descarte (COPY/DELETE), bem como inserir esse texto na especificação (INSERT/REPLACE/APPEND).

O menu **VIEW** permite definir a visibilidade dos componentes da especificação: tudo (SHOW ALL); apenas a parte em Estelle (ESTELLE); apenas as transições (TRANSITIONS); apenas o cabeçalho dos módulos e das subrotinas (HEADERS); apenas a parte de declarações (DECLARATIONS); tudo exceto a parte de declarações (STATEMENTS); somente os comandos de controle de fluxo (CONTROL FLOW); tudo menos comentários (NO COMMENTS).

O menu **OPTS** permite definir a tabulação (SET TABS), a forma de pesquisa de palavras (SET A=a / SET A<>a) e o modo de edição (SET TO BROWSE / SET TO EDITING).

4.2. EWSTRANS

Desenvolvido pela MARBEN com o auxílio da ferramenta "SYNTAX" (gerador de compiladores feito pela Universidade de Orleans), **EWSTRANS** transforma uma especificação em Estelle para

um código intermediário. Esse código intermediário pode ser utilizado para gerar um código orientado para implementação ou para simulação. Para executar EWSTRANS, digita-se o comando **goewstrans <spec>**, onde <spec> é o nome da especificação.

EWSTRANS desempenha as seguintes funções (Figura 4.3):

- analise léxico-sintático da especificação;
- geração e análise da tabela de símbolos e da árvore sintática (passo semântico);
- geração da listagem de análise e de referência cruzada;
- geração do código intermediário.

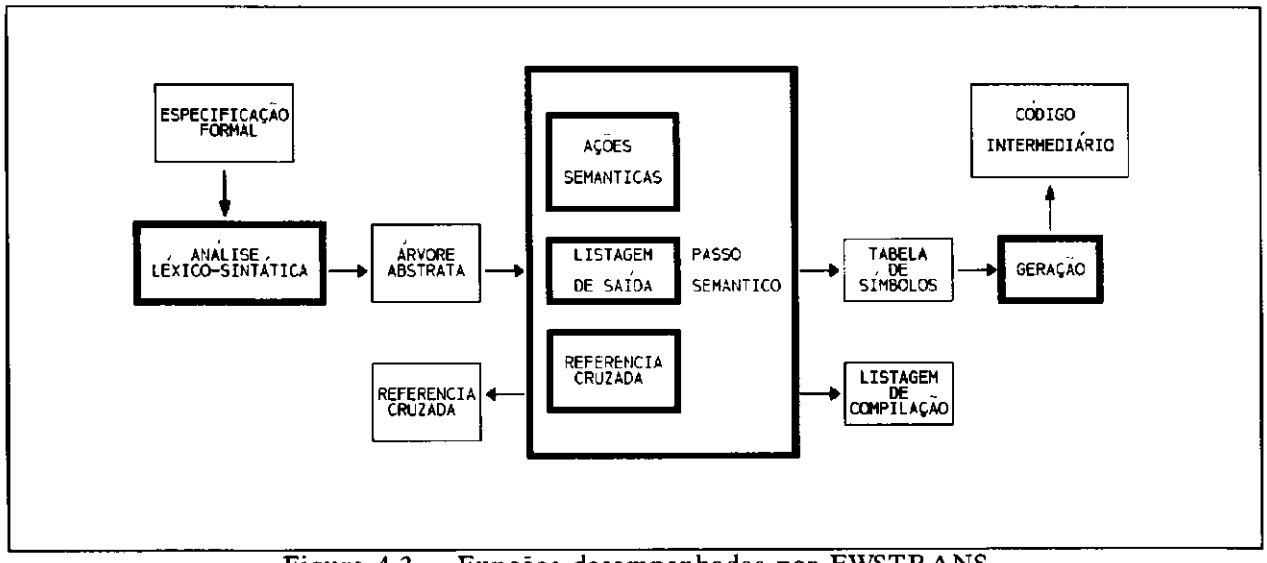

Figura 4.3 Funções desempenhadas por EWSTRANS

As principais restrições do EWSTRANS em relação à linguagem Estelle são:

- variáveis inteiras podem ter valores entre -2147483648 e 2147483647;
- seqüências e identificadores podem ter até 1024 caracteres;

- o tipo conjunto somente se aplica a tipos escalares, com valores entre 0 e 255;

- subrotinas do tipo **primitive** somente podem ser declaradas a nível global.

As mensagens de erro ou avisos podem ser de ordem léxico-sintática, de ordem semântica e de ordem interna ao EWSTRANS. O EWSTRANS, além de emitir avisos e erros, procura deduzir os erros de natureza léxico-sintática e corrigí-los automaticamente.

A listagem de referência cruzada contém os objetos da especificação (módulos, variáveis, canais, etc), os módulos, as transições e as estatísticas (quantidade de linhas, quantidade de módulos, etc).

EWSTRANS gera como saída os arquivos <spec>.stl.l e <spec>.if. O arquivo **<spec>.stl.l** contém a listagem de análise e de referência cruzada. O arquivo **<spec>.if** contém o código intermediário resultante.

4.3. EWSGEN

Desenvolvido pela VERILOG, **EWSGEN** gera um programa C a partir do código intermediário (<spec>.if). Pode ser gerado um programa orientado para implementação ou para simulação. EWSGEN realiza também a análise de referência.

O processo de geração do código C pode ser resumido da seguinte forma (Figura 4.4):

- carregamento do código intermediário (<spec>.if);

- criação de referências, através do reconhecimento e numeração de cada objeto, seguido pela construção de identificadores a serem utilizados no código gerado;

- geração de referências para estabelecer a ligação entre o código C gerado e o código intermediário, durante o processo de simulação (função ACT###);

- geração do código C;

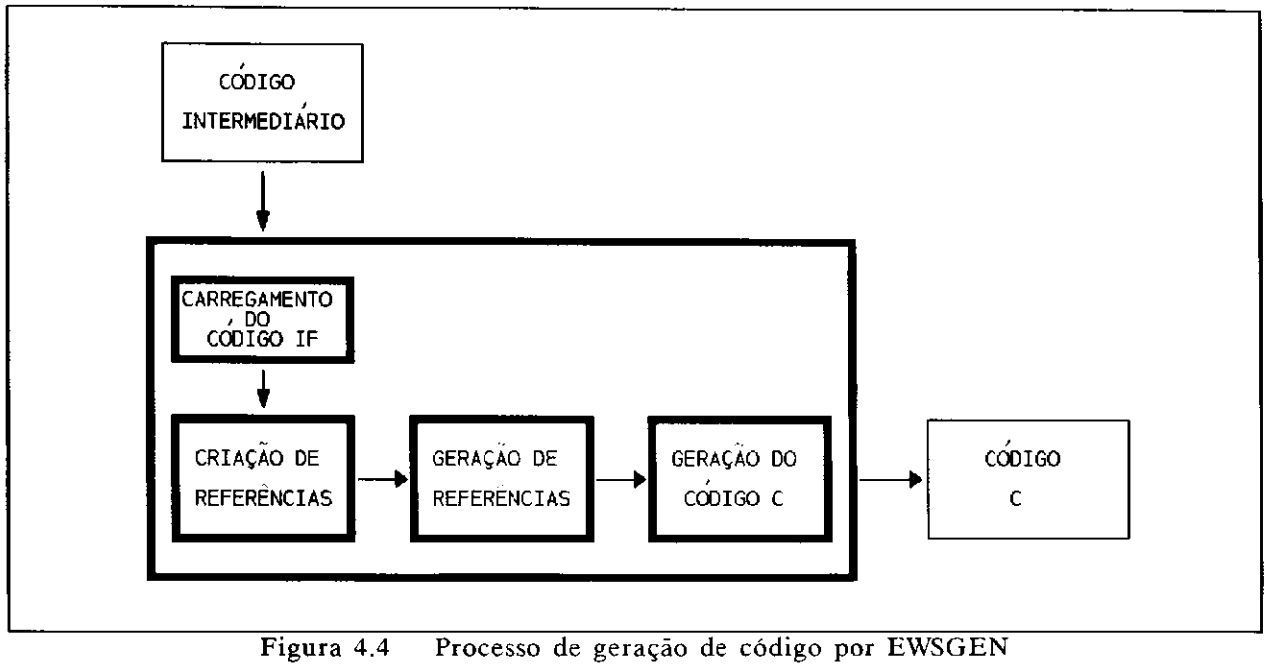

Figura 4.4 Processo de geração de código por EWSGEN

As estruturas de dados e subrotinas em Pascal são traduzidas para construções semelhantes em linguagem C.

Para gerar o código orientado para simulação, digita-se o comando **goewsgen <spec> -sim**. No caso do código orientado para implementação, digita-se o comando **goewsgen <spec> -imp**.

EWSGEN gera como saída os arquivos <spec>.c e <spec>.h. O arquivo **<spec>.c** contém o código C correspondente à especificação e o arquivo **<spec>.h** contém as definições da especificação.

O código para simulação contém um conjunto de funções de acesso (ACT###) a mais do que o código para implementação.

4.4. SIMULATOR/DEBUGGER

Desenvolvido pela E2S (interface do simulador) e VERILOG (núcleo do simulador), **SIMULATOR/DEBUGGER** permite simular uma especificação automaticamente ou interativamente. No modo interativo, o usuário seleciona uma transição para execução. No modo automático, uma transição é selecionada aleatoriamente para execução.

Um programa formado pelo código relativo à especificação e pelas rotinas do simulador é executado (comando **<spec>.simu <spec>**), oferecendo um ambiente completo de simulação. Esse programa é gerado da seguinte forma:

- geração do código intermediário a partir da especificação, através do comando **goewstrans <spec>;**
- geração do código C orientado para simulação, a partir do código intermediário, através do comando **goewsgen <spec> -sim** (nesse caso, o compilador C é automaticamente acionado para gerar o código objeto);
- compilação das subrotinas do tipo "primitive" através do comando **goewscompilprim <arquivo>;**
- ligação dos códigos objetos resultantes às rotinas de simulação através do comando **goewsmakesimu <spec>.**

A interface do simulador verifica a correção do comandos digitados pelo usuário e oferece recursos de ajuda "on-line" e "backtracking", permitindo o acesso do usuário ao núcleo do simulador. A interface do simulador foi gerada com auxílio da ferramenta MIRA.

A tela da interface do simulador é dividida em duas janelas: janela de conversação e janela de código fonte (Figura 4.5).

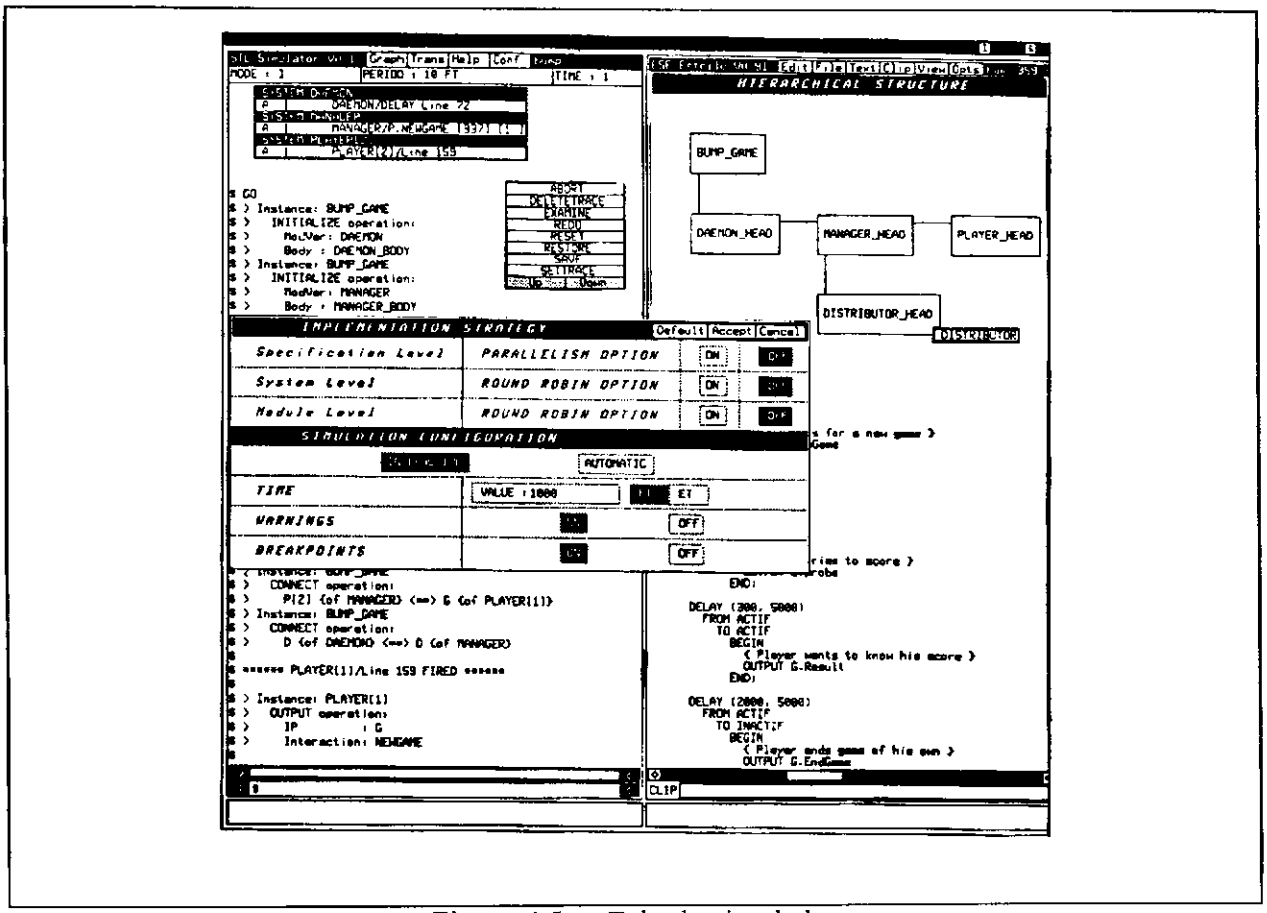

Figura 4.5 Tela do simulador

Na janela de conversação, o usuário informa todos os comandos ao simulador. Na janela de código fonte, é apresentado o último comando executado da especificação.

A janela de conversação é dividida em 3 partes:

- **cabeçalho**, onde a primeira linha mostra o nome da especificação e oferece 3 menus (GRAPH, TRANS e CONF), e a segunda linha contém a configuração da simulação;

- **janela de entrada e saída**, que fornece uma linha para a entrada de comandos, diversas linhas para a exibição dos traços da simulação (comandos fornecidos e seus resultados) e uma "scroll bar" para a visualização de todos os traços obtidos;

- **janela de mensagem**, que exibe as mensagens de erro.

O menu **GRAPH** do cabeçalho exibe o gráfico com a estrutura da especificação.

O menu **TRANS** exibe o conjunto de transições disparáveis, permitindo-se a seleção de uma delas para ser executada. Para cada transição disparável são identificados o módulo que contém a transição, o tipo de transição (primitiva de interação, "provided" e "delay") e o número da linha no código fonte.

O menu **CONF** permite configurar os seguintes parâmetros do simulador:

- **paralelismo da especificação**, para a simulação ou não do paralelismo assíncrono dentro de uma especificação;

- **seleção de módulos** ("system level"), para a utilização ou não do mecanismo "round-robin", que permite selecionar o módulo a ser executado;

- **seleção de transições** ("module level"), para a utilização ou não do mecanismo "round-robin", que permite selecionar a transição a ser executada;

- **modo de simulação**, que pode ser interativo, onde o usuário seleciona cada transição a ser disparada, ou automático, onde o simulador executa automaticamente até que o período de simulação termine;

- **período de simulação**, para a definição da unidade de tempo ou do número de transições a serem executadas e o valor do período (um número inteiro);

- **avisos em caso de erro**, que se for selecionado, exibe o resultado dos comandos INIT, RELEASE, CONNECT, DISCONNECT, ATTACH, DETACH, WHEN e OUTPUT:

- **ativação de "breakpoint"**, que se for selecionada, permite a interrupção da simulação a cada posição marcada ("breakpoint").

Os comandos disponíveis para a simulação são:

- **ABORT**, que permite abandonar o simulador;

- **DELTRACE**, que permite eliminar todos os traços estabelecidos para um módulo (DELTRACE <módulo>), para uma variável interna (DELTRACE <módulo> INTERNALVAR <variável>), para uma variável externa (DELTRACE <módulo> EXTERNALVAR <variável>) e para a variável de estado (DELTRACE <módulo> MAJORSTATE);

- **EXAMINE**, que permite examinar objetos Estelle, tais como, variáveis externas (EXAMINE <módulo> EXTERNALVAR <variável>), variáveis internas (EXAMINE <módulo> INTERNALVAR <variável>), pontos de interação do módulo (EXAMINE <módulo> IPCONNECTIONS), conteúdo de um ponto de interação ((EXAMINE <módulo> IPCONTENTS <ponto de interação> <primitiva> <parâmetro>) e variável de estado (EXAMINE <módulo> MAJORSTATE);

- **GO**, que é utilizado para inicializar a execução ou para reassumir a execução interrompida num "breakpoint";

- **REDO**, que permite repetir a execução de todas as transições armazenadas no período anterior;

- **RESET**, que reinicializa a simulação, retornando à configuração e ao estado iniciais;

- **RESTORE SCENARIO**, que executa automaticamente um cenário, previamente armazenado através do comando SAVE SCENARIO;

- **RESTORE INITIAL STATE**, que opera de forma semelhante a RESET, mas mantém a configuração vigente, os "breakpoints" e os traços;

- **SAVE CONFIGURATION**, que armazena a configuração vigente;

- **SAVE SCENARIO**, que armazena todas as seqüências executadas até o momento;

- **SETTRACE**, que permite fixar os traços para as variáveis externas (SETTRACE <módulo> EXTERNALVAR <variável>), para as variáveis internas (SETTRACE <módulo> INTERNALVAR <variável>) e para a variável de estado (SETTRACE <módulo> MAJORSTATE);

- **SETVALUE**, que permite alterar o valor de uma variável externa (SETVALUE <módulo> EXTERNALVAR <variável>) ou interna (SETVALUE <módulo> INTERNALVAR <variável>);

- **UNDDO**, que retorna ao estado anterior à execução da última transição;

- **USECONFIGURATION**, que utiliza a configuração armazenada pelo comando SAVE CONFIGURATION.

O simulador inicia suas atividades quando uma transição é selecionada para execução. Após a execução dessa transição, o status da especificação pode ser inspecionado (comando EXAMINE) e podem ser marcados pontos que interrompem a execução da especificação ("breakpoints"). O conteúdo das variáveis pode ser exibido sempre que for alterado (comando SETTRACE). Uma seqüência de execução pode ser repetida (comando RESTORE SCENARIO) quando armazenada (comando SAVE SCENARIO).

4.5. ESKIMO

ESKIMO (núcleo de implementação) é um conjunto de funções a ser utilizado pela implementação (Figura 4.5). Desenvolvido pela MARBEN, ESKIMO provê rotinas para a gerência de buffer, o escalonamento da execução dos módulos (núcleo de exploração) e a comunicação entre módulos-sistemas de uma mesma especificação, o que simplifica a tarefa de implementação.

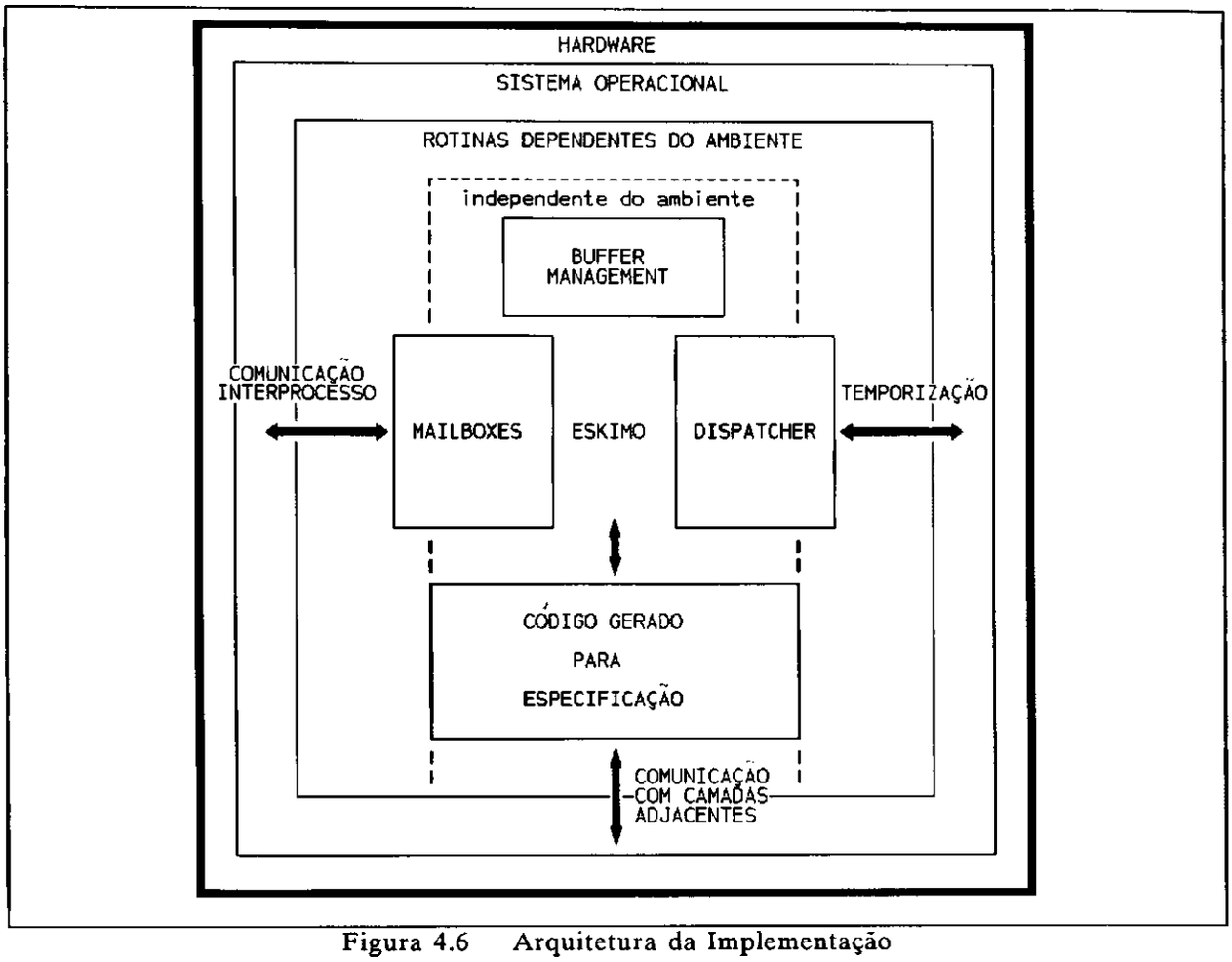

Figura 4.6 Arquitetura da Implementação

O <u>núcleo de exploração</u> se encarrega apenas da seleção do módulo a ser executado. A função Gub### (referente ao código C gerado a partir da especificação) seleciona a transição a ser executada. O algoritmo, responsável pela seleção do módulo a ser ativado, é baseado no princípio "round-robin". Os módulos pais

têm preferência em relação aos módulos filhos. O paralelismo, entre os módulos que envolve o atributo ("activity" ou "process"), é implementado dentro da função **_eeval** (DISPATCH.C). As rotinas do núcleo de exploração podem ser agrupadas da seguinte forma:

- rotinas para as declarações de constantes (GesCon), tipos (GESTYP e EWSIMPH) e macros (GesMac);

- rotinas de avaliação MakeListTrans, PuTT, PuTTDA, SaveDelay, GetListDelay, IsActiveDelay, Randomize, GetListTrans, GetListSpont, SortFrom, SortPrio, GetListDA, GetListDA_Version_Futur_a_tester, SortDA, IsInLst, AppWhen, CatList, CpyTT, MakeAnyVal, PutAnyVal, CpyZone, CreerEnum, InitEnum, ChercheEnum, GetEnum, EvalReset, Set_Init, Set_affect, AffectElem, Set_vaffect, set_add, set_sub, set_mul, set_in, set_egal, set_dif, set_inf, cmpzone, _getinter, err_affct, _Internal_Error, TestIndex, TestNul, TestInterval, cpbool, DivZero;

- rotinas que implementam as construções Estelle _attach, _connect, _detach, _output, _release, _freectx, whenget, _wheninter, _Xwhenlip, _getall, _getone, _Initall, _XMkCtx (ativado por mkctx), _Xwakectx, _rmctx, _xmkinter, _rminter, _Xmodchk (ou _modchk), _ipdisconnect, _moddisconnect;

- rotinas de temporização _runtimer, _gettic, _mkdy, _insdy, _setdelay, _rmdy, _rstdelay;

- rotinas do núcleo de exploração _etime, _eeval e _erun.

As rotinas para a gerência de buffer são independentes do sistema operacional, uma vez que a função padrão **calloc** é utilizada. São bastante úteis no auxílio à implementação dos protocolos das camadas do modelo OSI. São compostas basicamente por funções para alocação e liberação de memória, leitura e

gravação de dados, alteração do tamanho do buffer, concatenação de blocos, etc. As rotinas de gerência de buffer podem ser divididas em dois grupos:

- BUFMAP (bufnul, bufrst, _getdat, _reldat, bufget, bufrsx, _bufwlk, _ebmcpy, bufskx, bufrd, bufrds, bufwr, bufwrs, bufasm, buffrg, bufstl, bufspc, bufrem, bufrel, bufdeq, bufenq, bufmov, bufdup, bufgiv);

- BUFOSI (getdata, reldata, osigetb, osirelb, osispcb, ositlb, osiremb, osiasmb, osifrgb, osistxb, osiskpb, osiwrb, osirdb, osiexit, poolinit, pooldinit, poolminit, poolddinit, xosicopy, xosiscopy, xosipad, xosicomp, xosidump).

As facilidades para comunicação com outros processos são necessárias para a comunicação entre os sistemas de uma mesma especificação, pois o núcleo de exploração dá suporte somente às especificações que contenham um único módulo do tipo sistema. O modelo adotado para essa comunicação baseia-se no esquema de acesso a caixas postais ("mailboxes"). Cada caixa postal tem um nome e somente o seu dono pode ler o seu conteúdo. Qualquer outro processo apenas pode mandar informação para essa caixa postal. Para utilizar as facilidades oferecidas pelas caixas postais é necessário criar módulos que simulem outros sistemas.

As caixas postais podem ser implementadas por qualquer tipo de comunicação entre processos. No Unix-V, podem ser implementadas utilizando os seguintes mecanismos: "shared_memory", "message-queue", "named-pipes", etc. ESKIMO implementa as caixas postais utilizando "message-queue". As rotinas das caixas postais podem ser agrupadas da seguinte forma:

- rotinas para a declaração dos objetos ECMCON (constantes), ECMMAC (macros) e ECMTYP (estruturas de dados);

- rotinas que implementam as funções de acesso às caixas postais e que permitem a verificação de conteúdo (mxcontent), a leitura (mxhread e mxiread), a gravação (mxwrite), a abertura da caixa postal (mxiop e mxoop) e o fechamento da caixa postal (mxoclos e mxiclos).

ESKIMO implementa um mecanismo interno de gerência de memória, para reduzir a quantidade de chamadas ao sistema operacional. Dessa forma, páginas de memória são alocadas em uma única operação, através da função padrão **calloc**, formando um banco de memória (**memory pool**), que está sempre disponível para ser utilizado pelas estruturas de dados que implementam as construções Estelle (informações contextuais sobre as instâncias de módulos, filas de interações, lista de transições retardadas pelo comando "delay"). O acesso ao "memory pool" é realizado através das funções ppalloc, _ppfree, _ppinit e _ppsalloc.

A adaptação das rotinas de suporte ao ambiente operacional envolve a codificação das funções de temporização (cláusula DELAY) e a codificação das funções de comunicação entre processos do sistema operacional, utilizadas para implementação das caixas postais.

5. A Camada de Transporte

A camada de Transporte utiliza os serviços da camada de Rede ([CCIT 88c] e [CCIT 88d]) e oferece serviços à camada de Sessão ([CCIT 88e] e [CCIT 88f]). É a primeira camada fim-a-fim e tem a responsabilidade do transporte confiável de dados entre sistemas abertos, liberando as camadas superiores dessa função.

As primeiras tentativas de padronização da camada de transporte foram feitas pela ECMA (padrão ECMA-72 [ECMA 81]) e pelo CCITT (recomendação S.70 [CCIT 80]). As definições mais recentes e mais completas da camada de transporte foram feitas pela ISO [ISO 86 e ISO 88b] e pelo CCITT [CCIT 88a e CCIT 88b].

5.1. Serviço de Transporte

O serviço oferecido pela camada de transporte possui as seguintes características:

- significado fim-a-fim;
- possibilidade de seleção da qualidade de serviço;
- independência dos recursos de comunicação oferecidos pela camada de rede;
- transferência transparente de informações;
- endereçamento não ambíguo dos usuários da camada de transporte.

Os serviços da camada de transporte são oferecidos nos pontos de acesso ao serviço de transporte (TSAPs). O processo usuário (camada de Sessão) comunica-se com a camada de Transporte através da troca de primitivas de serviço (SP). Em resposta a uma SP, uma unidade de dados do protocolo de transporte (TPDU) é

enviada à entidade par. A recepção de uma TPDU provoca a emissão de uma SP para a camada de sessão.

SPs correlatas são diferenciadas pela qualificação. A qualificação **request** de uma SP indica uma solicitação do processo usuário. Em conseqüência, uma TPDU tipo **request** é enviada para a entidade par. Ao receber essa TPDU, a entidade par emite uma SP complementar, no caso qualificada por **indication**.

As outras qualificações complementares das SPs são response e confirm. A qualificação **response** indica uma resposta do processo usuário par. Em conseqüência, uma TPDU do tipo **confirm** é enviada para a entidade, provocando a emissão de uma SP complementar **confirm**. A Tabela 5.1 resume as SPs oferecidas pela Camada de Transporte.

SPs		PARÂMETROS
T-CONNECT	request indication	Endereço do Chamado Endereço do Chamador Opção de Dados Expressos Qualidade de Serviço Dados do Usuário
T-CONNECT	response confirm	Endereço do Respondedor Opção de Dados Expressos Qualidade de Serviço Dados do Usuário
T-DATA	request indication	Dados do Usuário
T-EXPEDITED-DATA	request indication	Dados do Usuário
T-DISCONNECT	request	Dados do Usuário
T-DISCONNECT	indication	Razão da Desconexão Dados do Usuário

Tabela 5.1 SPs da camada de Transporte

De acordo com o diagrama da Figura 5.1, as seguintes funções são desempenhadas pelas SPs: estabelecimento de conexões, transferência de dados e encerramento de conexões.

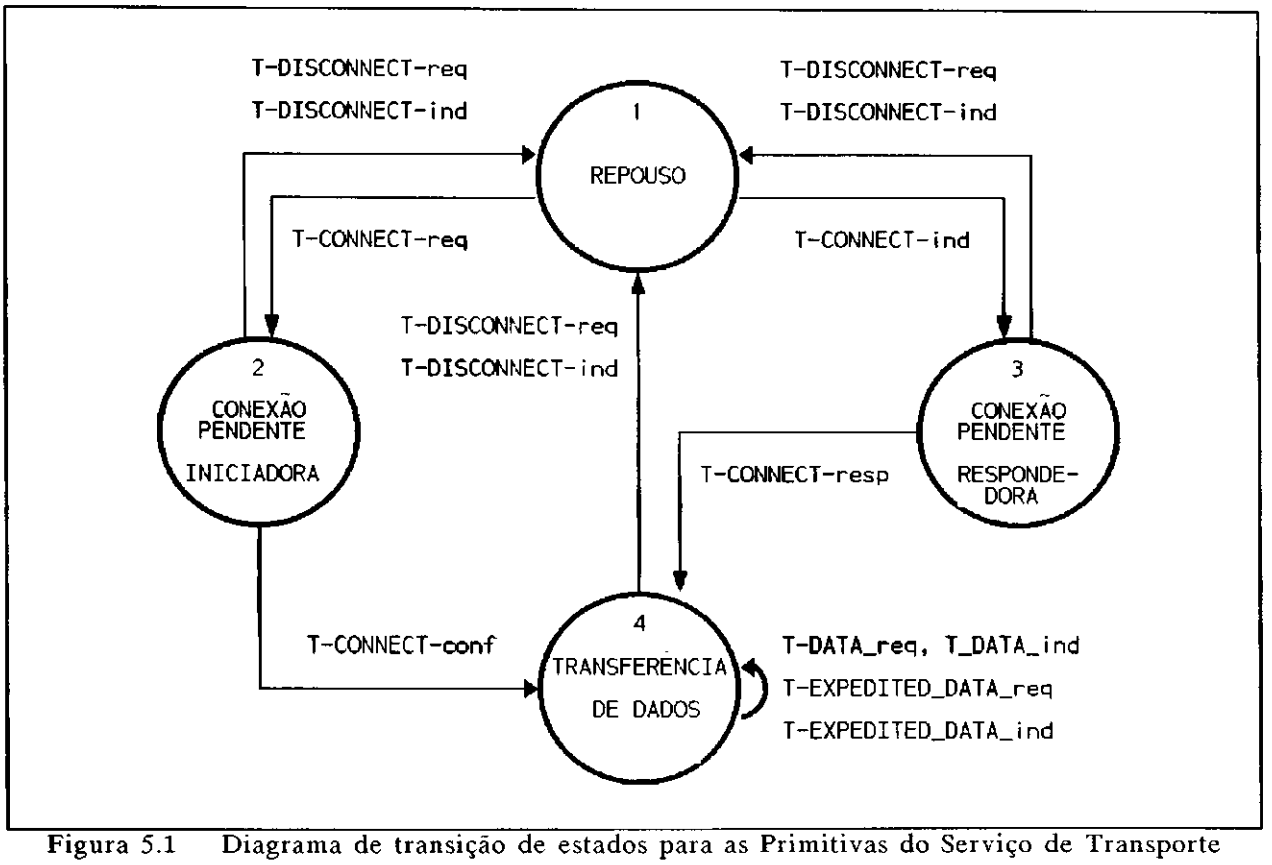

Figura 5.1 Diagrama de transição de estados para as Primitivas do Serviço de Transporte

5.1.1. Estabelecimento de Conexões

As SPs **T-CONNECT** são utilizadas para estabelecer uma conexão de transporte (TC). A entidade de transporte que deseja estabelecer uma conexão é chamada iniciadora e a entidade destinatária dessa conexão é denominada respondedora.

No estabelecimento bem sucedido de uma TC, o usuário iniciador emite uma SP **T-CONNECT-request** à entidade iniciadora, que por sua vez envia uma TPDU **CR** à entidade respondedora. Ao receber essa TPDU, essa entidade emite uma SP **T-CONNECT-indication**. Ao receber essa SP, esse usuário emite uma SP **T-**

CONNECT-response à entidade respondedora, que por sua vez envia uma TPDU **CC** à entidade iniciadora. Ao receber essa TPDU, essa entidade emite uma SP **T-CONNECT-confirm** ao usuário iniciador, terminando assim a fase de estabelecimento de uma TC.

Na figura 5.2, além do comportamento descrito acima, estão ilustrados os casos de rejeição ao pedido de estabelecimento de uma TC.

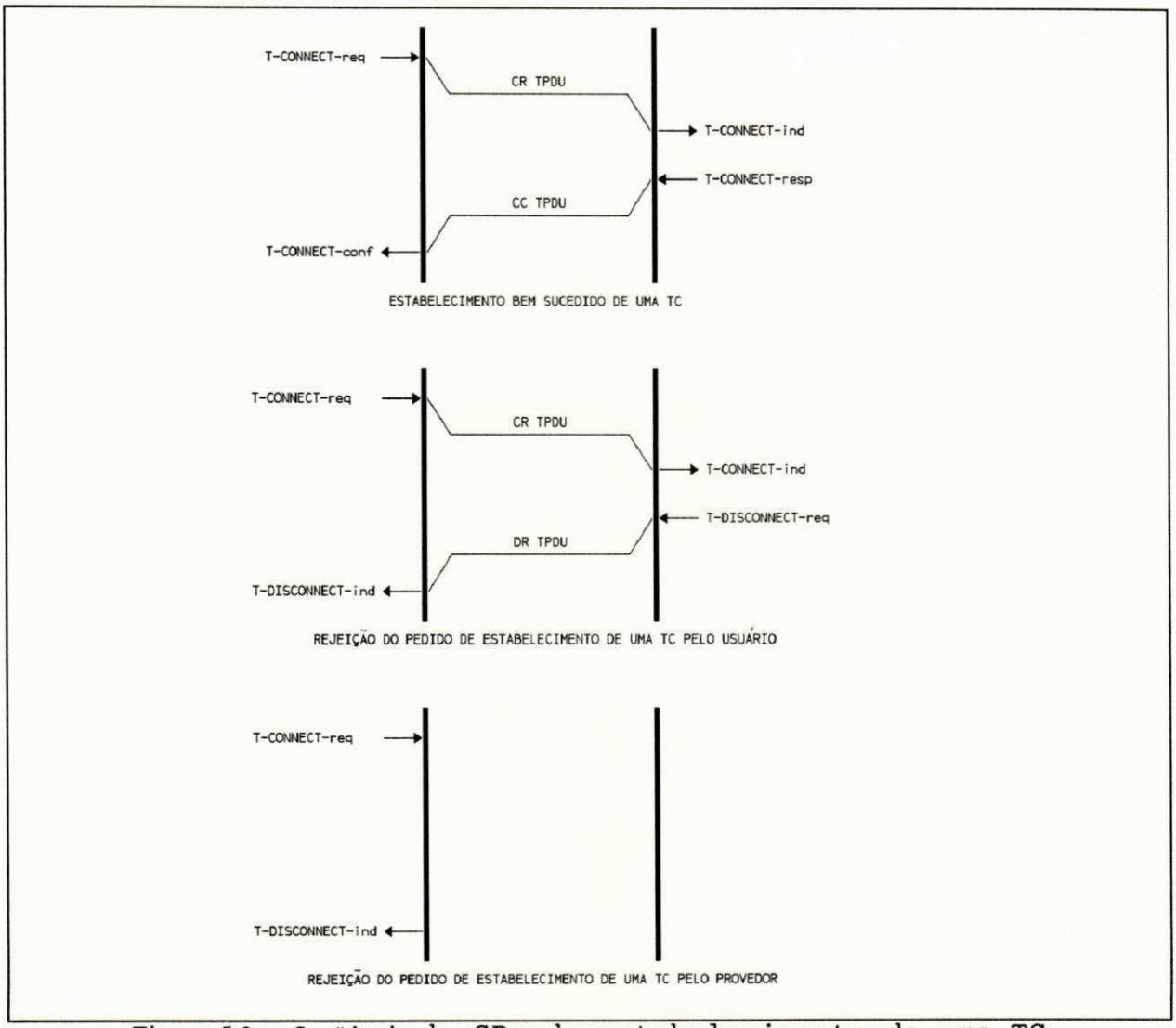

Figura 5.2 Seqüência de SPs de estabelecimento de uma TC

Em relação aos parâmetros das SPs T-CONNECT, as características do endereço de transporte (endereço do chamado,

endereço do chamador e endereço do respondedor) não são definidas nas recomendações do CCITT. Uma sugestão seria adotar uma estrutura hierárquica análoga à estrutura proposta na recomendação X.121 [CCIT 88g] para endereços de rede.

Os parâmetros opção de dados expressos e qualidade de serviço são negociados entre as entidades usuárias. O parâmetro **opção de dados expressos** indica a utilização do serviço de transferência de dados expressos.

O parâmetro **qualidade de serviço** é composto por vários itens. Podemos agrupá-los em : parâmetros relativos à performance e parâmetros relativos à confiabilidade.

Os parâmetros de qualidade de serviço relativos à performance são:

- **atraso no estabelecimento da TC** (TC establishment delay), que é o tempo máximo entre a emissão de uma primitiva T-CONNECT-request e a recepção de uma T-CONNECT-confirm, pelo usuário iniciador da conexão;

- **vazão** (throughput), sendo que a especificação desse parâmetro consiste na definição de um valor máximo e de um valor médio para cada sentido da transmissão;

- **atraso de trânsito** (transit delay), que é o tempo entre a emissão de uma primitiva T-DATA-request e a chegada da T-DATA-indication ao usuário par. Deve ser definido um valor máximo e um valor médio para cada sentido da transmissão;

- **atraso no encerramento da TC** (TC release delay), que é o tempo máximo entre a emissão de uma primitiva T-DISCONNECT-request e a liberação da conexão pelo usuário par. Deve ser definido um valor para cada sentido da transmissão.

Na monitoração dos parâmetros **atraso de trânsito** e **vazão**, somente as unidades de dados de serviço de transporte (TSDUs) transferidas corretamente são consideradas.

Os parâmetros de qualidade de serviço relativos à confiabilidade são:

- **probabilidade de falha no estabelecimento da TC** (TC establishment failure probability), que é o percentual de falhas no estabelecimento de uma conexão, medido num período de amostragem;

- **taxa de erro residual** (residual error rate), que é a taxa de TSDUs transmitidas erroneamente, medida num período de amostragem;

- **probabilidade de falha na transferência** (transfer failure probability), que é o percentual de falhas de transferência, medido num período de amostragem. A falha de transferência indica uma performance inferior à mínima aceitável;

- **probabilidade de falha no encerramento da TC** (TC release failure probability), que é o percentual de falhas nas tentativas de encerramento de uma TC;

- **proteção da TC** (TC protection), que indica o desejo de se proteger (ou não) os dados do usuário da TC contra monitoração ou gravação;

- **prioridade da TC** (TC priority), que indica a importância da TC em caso de degradação da qualidade dos serviços, ou em caso da necessidade de liberação da conexão para a recuperação dos recursos alocados.

O parâmetro **dados do usuário** permite transportar de 1 a 32 octetos.

5.1.2. Transferência de dados

A camada de transporte provê serviços que permitem a transmissão duplex de TSDUs. Existem dois tipos de serviço: transmissão de dados normais e transmissão de dados expressos.

Na transmissão de dados normais é garantida a ordem e a integridade das TSDUs. A SPs T-DATA são utilizadas para transmitir dados normais. A SP **T-DATA-request**, emitida pelo usuário, resulta na geração de TPDUs **DT**. A partir das TPDUs DT recebidas, é composta uma SP **T-DATA-indication**. O parâmetro **dados do usuário** corresponde a uma TSDU não nulo.

Na Figura 5.3 estão ilustradas as seqüências de SPs para a fase de transferência de dados.

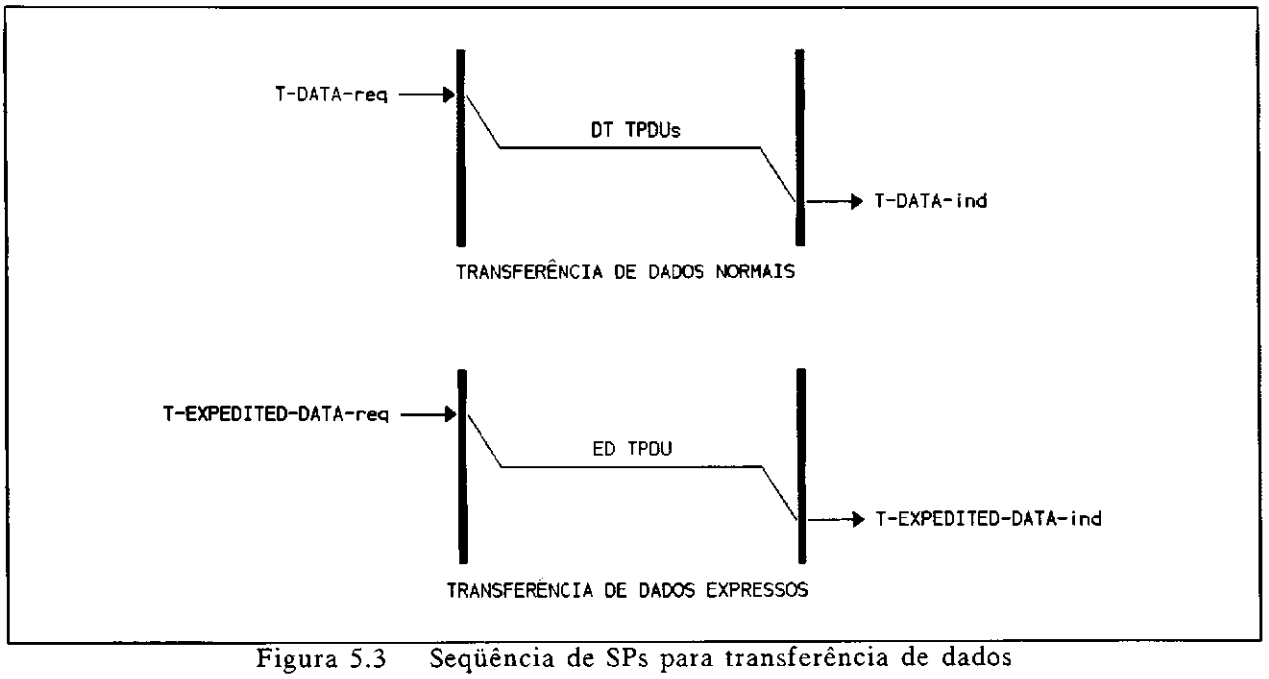

Figura 5.3 Seqüência de SPs para transferência de dados

Os dados expressos tem prioridade na transmissão e estão sujeitos a controle de fluxo e qualidade de serviço diferentes. A quantidade de dados transmitida no parâmetro **dados do usuário** da SP T-EXPEDITED-DATA está limitada a 16 octetos. A SP **T-EXPEDITED-DATA-request** é mapeada numa TPDU **ED** e, a partir dessa TPDU, a

entidade par gera uma SP **T-EXPEDITED-DATA-indication** na entidade par.

5.1.3. Encerramento de conexões

O pedido de encerramento de uma TC pode ser feito a qualquer momento e esse pedido não pode ser rejeitado. A entidade de transporte não garante a entrega de dados, uma vez iniciado o procedimento de encerramento da TC.

O encerramento de uma TC pode ser iniciada :

- por um ou ambos usuários da camada ou pela própria camada, liberando a TC existente;

- por um ou ambos usuários da camada ou pela própria camada, rejeitando um pedido de estabelecimento da TC.

As SPs T-DISCONNECT são utilizadas para liberar uma TC. A SP **T-DISCONNECT-request** é mapeada numa TPDU **DR** e, a partir da recepção dessa TPDU, a entidade par gera uma SP **T-DISCONNECT-indication**.

Na Figura 5.4 estão ilustradas as seqüências de SPs para a fase de encerramento de uma TC.

As SPs T-DISCONNECT possuem dois parâmetros. O parâmetro **dados do usuário** pode transportar entre 1 e 64 octetos. O parâmetro **Razão da desconexão** é fornecido se a desconexão foi iniciada pela camada.

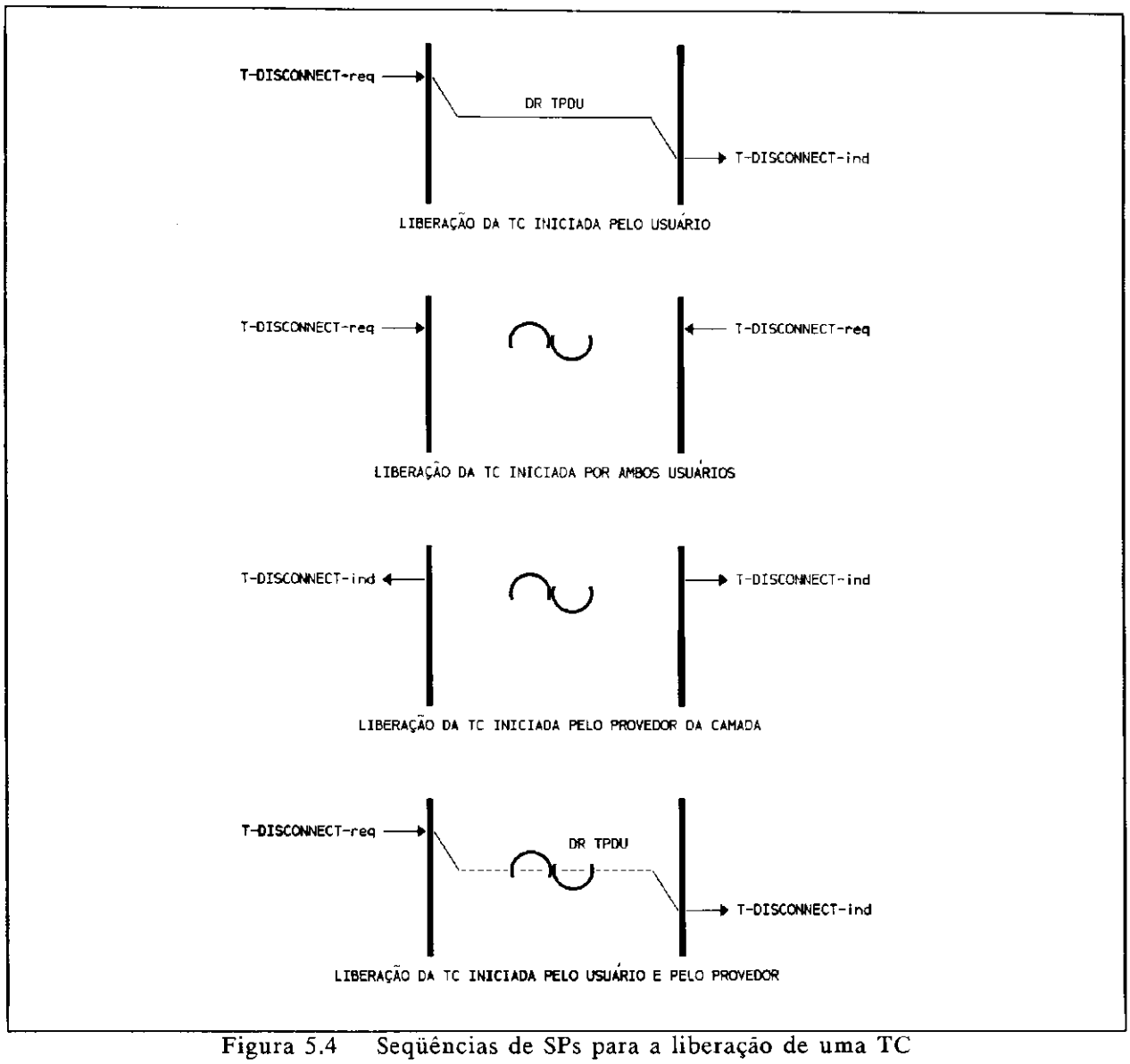

Figura 5.4 Seqüências de SPs para a liberação de uma TC

5.2. Protocolo de Transporte

O protocolo de transporte provê mecanismos para a troca de dados entre os processos usuários, liberando-os da tarefa de gerenciamento das facilidades de comunicação. Os dados do usuário são encapsulados numa TPDU, que contém também informações de controle.

A funcionalidade do protocolo de transporte é decorrente da natureza da camada de rede, dos requisitos da aplicação e das

características do usuário. Em função dessa funcionalidade, foram definidas 5 classes de protocolo de transporte: classe 0 (classe simples), classe 1 (classe com recuperação básica de erros), classe 2 (classe com multiplexação), classe 3 (classe com recuperação de erro e multiplexação) e classe 4 (classe com detecção e recuperação de erro).

As principais funções oferecidas pelo protocolo de transporte são:

- associação entre TCs e NCs (todas as classes), criando (N-CONNECT) ou liberando (N-DISCONNECT) a conexão de rede;

- transferência de TPDUs, utilizando as SPs N-DATA (todas as classes) e N-EXPEDITED-DATA (classe 1);

- segmentação e montagem (todas as classes), permitindo segmentar uma TSDU (dados do usuário) em várias TPDUs DT;

- concatenação e separação (exceto classe 0), permitindo concatenar diversas TPDUs no campo **dados do usuário** da SP N-DATA;

- estabelecimento de uma TC sobre uma NC existente (todas as classes), sendo que a SP N-DATA é utilizada para transportar uma TPDU CR ou uma TPDU CC;

- recusa de uma conexão, enviando uma TPDU ER (todas as classes) ou uma TPDU DR (exceto classe 0);

- liberação de uma conexão, que pode ser implícita (classe 0), onde a NC é liberada conjuntamente com a TC (N-DISCONNECT), explícita (exceto classe 0), onde apenas a TC é liberada (através de uma TPDU DR e de uma TPDU DC);

- liberação da TC por motivo de erro (classe 0 e 2);

- associação de TPDUs às TCs (todas as classes);

- numeração da TPDU DT (classes 1,2,3 e 4) para as funções de controle de fluxo, recuperação de erro e reordenação de TPDUs;

- transferência de dados expressos (classes 1,2,3 e 4);

- atribuição de uma outra NC para a TC (classes 1 e 3), permitindo recuperação após uma desconexão sinalizada;

- retenção da TPDU até o recebimento de uma TPDU AK (classes 1, 3 e 4);

- ressincronização (classes 1 e 3);

- multiplexação e desmultiplexação (classes 2,3 e 4);

- controle de fluxo explícito (classes 2, 3 e 4);

- utilização de checksum para detectar erros de transmissão (classe 4);

- congelamento de referências (classes 1, 3 e 4);

- retransmissão de TPDUs em caso de time-out do temporizador (classe 4);

- reordenação de TPDUs ED e de TPDUs DT fora de seqüência (classe 4);

- controle de inatividade para permitir o tratamento de desconexão não sinalizada (classe 4);

- tratamento de erros de protocolo;

- dispersão e recombinação (classe 4), permitindo multiplexar uma TC em várias NCs.

Durante a fase de abertura de uma conexão, diversas opções são negociadas:

- classe de serviços;

- parâmetros de qualidade de serviço;

- comprimento das TPDUs;

- opções dados expressos (exceto classe 0), utilização de N-DATA-ACKNOWLEDGE (classe 1), utilização de N-EXPEDITED-DATA (classe 1), utilização de checksum (classe 4);

- opções formato normal ou estendido (classes 2, 3 e 4) e controle de fluxo explícito (classe 2).

Duas entidades da camada de Transporte comunicam-se através da troca de TPDUs. As TPDUs são mapeadas para o campo dados do usuário da SP N_DATA da camada de rede. A estrutura de uma TPDU (Figura 5.5) é composta pelo **cabeçalho** e pelo campo **dados do usuário**. O tamanho de uma TPDU é fixo e negociado na fase de estabelecimento de uma conexão. O tamanho do cabeçalho é indicado pelo campo Length Indicator (**LI**). A parte fixa do cabeçalho é composta por campos com tamanhos conhecidos. A parte variável do cabeçalho é composta por estruturas com as seguintes características: identificador do parâmetro (1 octeto), tamanho do parâmetro (1 octeto) e conteúdo do parâmetro (n octetos).

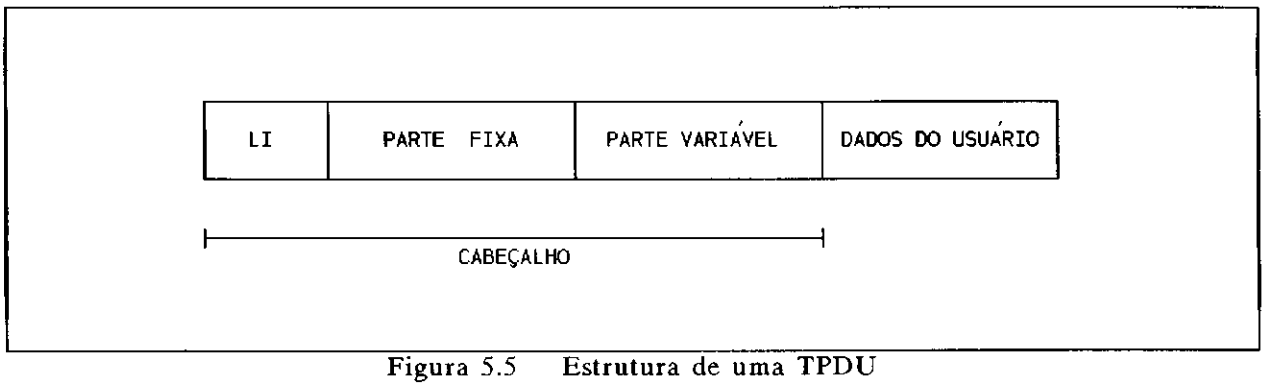

Figura 5.5 Estrutura de uma TPDU

Em relação à função de cada TPDU, **CR** e **CC** são utilizadas na fase de abertura de conexões, **DR** e **DC** são utilizadas na fase de encerramento de conexões, **DT**, **ED**, **AK** e **EA** são utilizadas na fase

de transferência de dados, **RJ** permite rejeitar uma série de **DTs** e **ER** indica a detecção de erro de protocolo.

Em relação aos parâmetros da parte fixa, o parâmetro **CDT** indica a quantidade de créditos concedidos, os parâmetros **DST-REF** e **SRC-REF** identificam a referência dada a cada TC, os parâmetros **TPDU_NR** e **ED-TPDU-NR** contém o número de seqüência de uma DT e ED respectivamente, o parâmetro **EOT** indica o fim de uma TSDU e o parâmetro **YR-TU-NR** contém o número de seqüência da próxima DT esperada.

Na figura 5.6 estão representadas as estruturas das seguintes TPDUs:

- Connection Request (**CR**) e Connection Confirm (**CC**), onde o tamanho máximo é 128 octetos. A parte variável pode conter os parâmetros identificador de TSAP, tamanho da TPDU, versão, parâmetro de segurança, checksum, seleção adicional de opções, classes alternativas de protocolo, tempo máximo de reconhecimento (acknowledge), vazão, taxa de erro residual, prioridade, atraso de trânsito e tempo para atribuição de uma outra NC. O parâmetro **dados do usuário** pode transportar até 32 octetos (exceto classe 0);

- Disconnection Request (**DR**), onde a parte variável pode conter os parâmetros informações adicionais sobre a liberação da conexão e checksum. O tamanho máximo da TPDU é de 128 octetos e o parâmetro **dados do usuário** pode transportar até 64 octetos (exceto classe 0);

- Disconnection Confirm (**DC**), onde a parte variável pode conter o parâmetro checksum (essa TPDU não é utilizada na classe 0);

- Data (**DT**), onde a parte variável pode conter o parâmetro checksum;

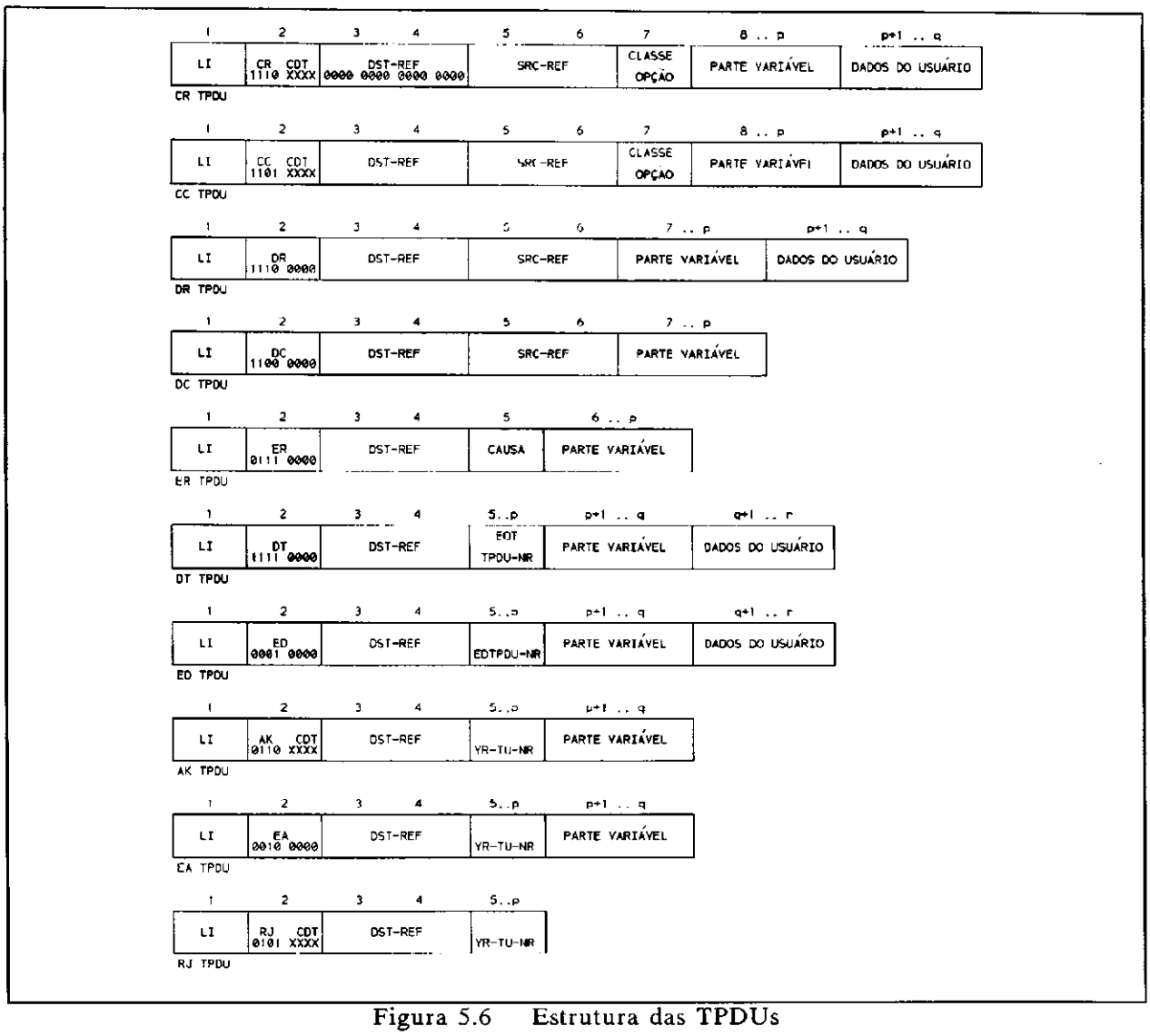

Figura 5.6 Estrutura das TPDUs

- Expedited Data (**ED**), onde a parte variável pode conter o parâmetro checksum e o parâmetro dados do usuário pode transportar até 16 octetos (essa TPDU não é utilizada na classe 0);

- Acknowledgement (AK), onde a parte variável pode conter os parâmetros checksum, número de subseqüência e confirmação de controle de fluxo (essa TPDU não é utilizada na classe 0);

- Expedited Data Acknowledgement (**EA**), onde a parte variável contém o parâmetro checksum (essa TPDU não é utilizada na classe 0);

- Error (**ER**), onde a parte variável contém os parâmetros checksum e TPDU inválido;

- Reject (**RJ**), que é utilizada nas classes 1 e 3.

5.2.1. Classes de Protocolo

As classes de protocolo de transporte (Figura 5.7) foram definidas com base na existência de uma classificação dos serviços de rede. De acordo com a qualidade, a camada de rede pode ser classificada em:

- tipo A, onde as conexões de rede possuem taxa aceitável de erro residual e falha sinalizada;

- tipo B, onde as conexões de rede possuem taxa aceitável de erro residual, mas taxa inaceitável de falha sinalizada;

- tipo C, onde as conexões de rede possuem taxa inaceitável de erro.

Apesar da diferenciação em classes de protocolo, a especificação é comum quanto às primitivas de serviço e às estruturas de dados. Cada classe implementa um conjunto de funções.

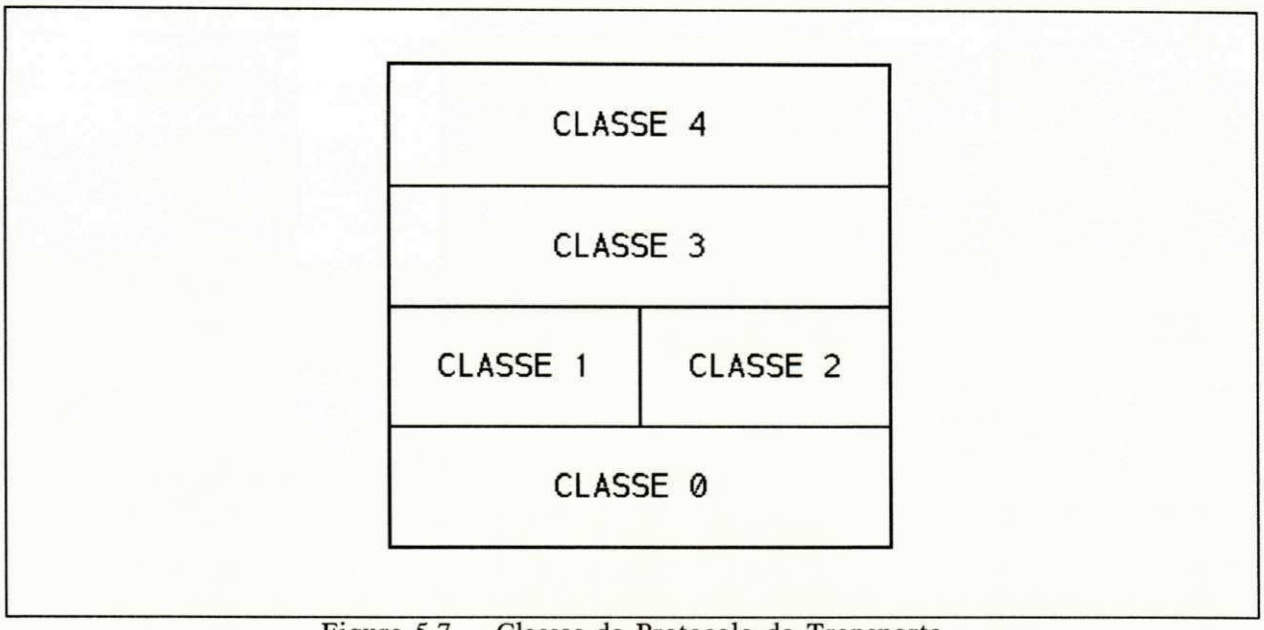

Figura 5.7 Classes de Protocolo de Transporte

5.2.1.1. Classe 0 - Classe simples

Implementa a funcionalidade mínima, compatível com a recomendação CCITT T.70 para terminais TELETEXT. O tempo de vida de uma TC é o mesmo que o de uma NC. O mecanismo de controle de fluxo é baseado no serviço da camada de rede. Utiliza redes do tipo A.

Na fase de estabelecimento de conexão, somente os parâmetros identificador do TSAP (**TSAP-ID**) e tamanho de TPDU são negociados. O tamanho padrão de uma TPDU é de 128 octetos, podendo ser negociados valores, tais como, 256, 512, 1024 e 2048.

Na ocorrência de um erro sinalizado, a função de desconexão implícita é utilizada e o usuário é informado do problema.

5.2.1.2. Classe 1 - Classe de recuperação de erros básicos

A classe 1 foi projetada para usar redes do tipo B, podendo recuperar-se a partir de erros sinalizados, tais como, desconexão ou reinicialização (reset) de uma NC. Oferece funções para

transferência de dados expressos e controle de fluxo, baseadas nos serviços da camada de rede.

Permite enviar dados no estabelecimento de conexões, reutilizar a mesma NC para as próximas TCs, ou seja, o tempo de vida das TCs é independente do tempo de vida das NCs, e segmentar uma TSDU em várias TPDUs DT.

O procedimento de ressincronização permite fazer a TC voltar ao normal após uma falha. A entidade iniciadora é responsável pelo restabelecimento da NC, pela retransmissão de uma ED ou de uma DT (que esteja aguardando uma confirmação) e pela retransmissão de uma RJ com o número da próxima DT.

A liberação da conexão é efetuada utilizando-se uma DR e DC.

5.2.1.3. Classe 2 - Classe de multiplexação

Projetada para ser utilizada em associação com uma conexão de rede do tipo A, o protocolo de Transporte classe 2 tem as seguintes características:

- multiplexação de várias conexões de transporte numa única conexão de rede;
- mecanismos explícitos de encerramento de conexões;
- controle de fluxo e dados urgentes opcionais;
- nenhuma função de recuperação e detecção de erros, sendo a conexão liberada em caso de desconexão ou reinicialização da NC.

O uso de controle de fluxo proporciona controle sobre eventuais congestionamentos através de mecanismos de crédito. A redução de créditos não é permitida.

Cada **ED** deve ser confirmada com uma **EA**, tendo, portanto, um controle de fluxo diferenciado em relação aos dados normais.

5.2.1.4. Classe 3-Classe de recuperação de erros e multiplexação

A classe 3 foi projetada para ser utilizada em associação com conexões de rede do tipo B. Essa classe estende a funcionalidade da classe 2, adicionando a habilidade para recuperação da TC, após uma falha sinalizada pelo nível de rede, sem envolver o usuário do serviço de transporte.

A entidade de transporte retém as **DTs** (dados) e as **EDs** (dados expressos) até que ela receba uma confirmação (**AK** ou **EA**). O recebimento de uma **RJ** indica quais as TPDUs que devem ser repetidas. Os campos CDT e YR-TU-NR da RJ podem também reduzir a quantidade de créditos concedidos.

Uma **ED** é transmitida com um número de seqüência diferente (ED-TPDU-NR), sendo empacotada dentro do campo dados do usuário da SP N-DATA. Uma confirmação **EA** será sempre enviada após a recepção de uma ED. A SP T-EXPEDITED-DATA-indication só será emitida caso o valor do campo ED-TPDU-NR seja diferente dos últimos recebidos.

A liberação das conexões de transporte é feita de forma explícita, utilizando **DR** e **DC**.

5.2.1.5. Classe 4 - Classe de detecção e recuperação de erros

Esta classe, projetada para ser utilizada com redes do tipo C, provê a funcionalidade da classe 3 mais a capacidade de detectar erros e se recuperar após esses erros. Dentre os erros, podemos citar a perda de TPDUs, dados fora de seqüência, duplicação de TPDUs, TPDUs danificadas, etc. A detecção de erros

é feita pelo uso de mecanismos de time-out, numeração de TPDUs e checksum (função opcional).

O mecanismo de time-out baseia-se na utilização de uma série de temporizadores: temporizadores para tempo de vida de NSDUs ($\mathbf{M_{lr}}$ e $\mathbf{M_{rl}}$), temporizadores para retardo de trânsito ($\mathbf{E_{lr}}$ e $\mathbf{E_{rl}}$), temporizadores para controle de reconhecimento ($\mathbf{A_l}$ e $\mathbf{A_r}$), temporizador para retransmissão ($\mathbf{T_l}$), temporizador de persistência (**R**), temporizador de reutilização de referências (**L**), temporizador de inatividade (**I**) e temporizador de freqüência de atualização das janelas (**W**).

Na classe 4, uma conexão de transporte pode ser mapeada em várias conexões de rede (splitting). Isto visa conseguir proteção adicional contra falhas e obter uma vazão maior. A numeração das TPDUs ED e DT permite que os dados sejam reordenados.

Na fase de transferência de dados, as TPDUs com dados possuem tempo de vida limitado. Dados não confirmados são retransmitidos após o vencimento do prazo do temporizador T_l, por um período determinado pelo temporizador R. Dados recebidos fora de seqüência são armazenados até a obtenção da seqüência correta. Dados enviados são armazenados até a recepção de confirmação. A quantidade de créditos pode ser reduzida.

O dados expressos são enviados ao usuário logo após a chegada de uma ED, caso o valor do campo ED-TPDU-NR (numeração) seja diferente. A entidade emissora não enviará nenhum dado expresso até a confirmação.

Existe um temporizador de inatividade (**I**), que libera a conexão quando não há transmissão de TPDUs.

Na fase de desconexão, as referências de conexão (DST-REF e SRC-REF) são congeladas, podendo a referência ser reutilizada somente após um tempo (**L**).

5.3. Especificação Informal do Protocolo de Transporte Classe 2

O protocolo de Transporte, escolhido para implementação neste trabalho, foi o classe 2 (Figura 5.8). Dessa forma, alguns detalhes adicionais são apresentados nesta seção.

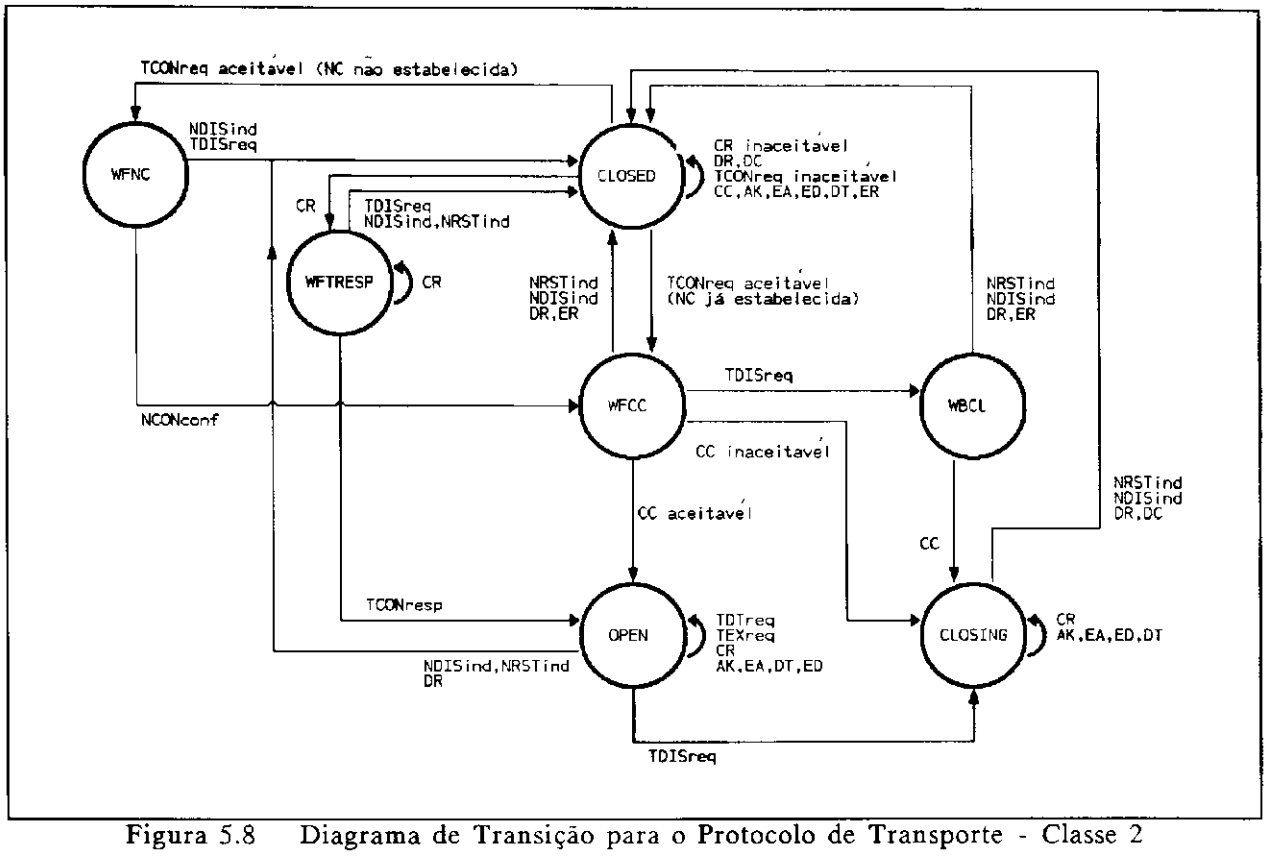

Figura 5.8 Diagrama de Transição para o Protocolo de Transporte - Classe 2

Durante a fase de estabelecimento de conexão, a entidade de transporte do lado iniciador mapeia a SP **T-CONNECT-request** numa **TPDU CR**. Caso exista uma conexão de rede já estabelecida, que possa atender à qualidade de serviço requerida, e for possível a multiplexação, esta NC pode ser utilizada. Se nenhuma NC existente puder ser utilizada, uma nova NC deve ser criada (emissão de **N-CONNECT-request** e recepção de **N-CONNECT-confirm**). Se nenhuma NC puder ser utilizada, uma SP **T-DISCONNECT-indication** é emitida ao usuário.

Ao receber uma **CR**, o lado respondedor envia uma SP **T-CONNECT-indication**. O usuário pode aceitar o pedido de conexão (emitindo T-CONNECT-response) ou rejeitá-lo (emitindo T-DISCONNECT-request). A recepção de **CRs** duplicadas deve ser ignorada, se o iniciador indicar a classe 4 como a classe preferencial.

Durante a abertura de uma TC, é feito uma série de negociações a respeito das características da conexão: classe de protocolo, tamanho das TPDUs, opções de dados expressos, formato normal ou estendido, uso de controle de fluxo explícito e qualidade de serviço.

Um temporizador opcional **TS1** é associado ao envio de uma **CR**. Caso vença o prazo do temporizador TS1 e nenhuma **CC** ou **DR** for recebida, a TC deve ser liberada.

A multiplexação de várias TCs numa única NC permite otimizar a taxa de utilização dessa NC. Diversas TPDUs do tipo AK, EA, RJ, ER e DC podem ser concatenadas no campo **dados do usuário** de uma SP **N-DATA**, desde que sejam oriundas de diferentes TCs. Entretanto, o campo dados do usuário da SP N-DATA pode transportar as TPDUs anteriores, combinadas com apenas uma TPDU do tipo CR, DR, CC, DT e ED.

Uma vez estabelecida a TC, os usuários podem trocar dados normais ou dados expressos. A SP **T-DATA-request** é mapeada em uma ou mais **DTs**. Cada DT é numerada, permitindo o controle de fluxo.

O controle de fluxo é opcional e negociado entre as entidades de transporte. O uso de controle de fluxo, aplicado aos dados normais, proporciona o controle sobre eventuais congestionamentos. Um mecanismo de crédito é utilizado, permitindo ao receptor informar ao emissor a quantidade de dados que pode ser recebida. A quantidade de créditos é incrementada, com base nas informações fornecidas pelos parâmetros YR-TU-NR e

CDT da TPDU **AK**. A entidade somente pode transmitir ou receber uma **DT**, cujo número de seqüência esteja dentro da janela. A janela de transmissão é alterada com a recepção de uma **AK** e a emissão de uma **DT**. A redução de créditos não é permitida.

A utilização de dados expressos é uma opção do usuário. A SP T-EXPEDITED-DATA-request é mapeada em uma ED. Cada ED deve ser confirmada com uma EA, antes do envio da próxima ED.

O protocolo de transporte classe 2 não possui nenhuma função de recuperação e detecção de erros. Por exemplo, a TC é liberada em caso de desconexão ou reinicialização da NC. A TC também deve ser liberada em caso de erros de protocolo, tais como, número de seqüência fora da janela, tentativa de redução de créditos, recepção de dados expressos quando a opção não estiver selecionada, recepção de TPDUs que não possam ser decodificadas ou que contenham parâmetros inválidos, etc. Nesses casos, o protocolo pode também desconectar a NC, reinicializar a NC, ou enviar uma TPDU ER contendo o erro e liberar a TC.

Alguns erros podem ser desprezados, tais como recepção de primitivas inválidas e recepção de TPDUs não permitidas para o estado do protocolo.

Os mecanismos de encerramento de TC são explícitos, com a utilização de DR e DC. Após o envio de uma **DR**, o temporizador opcional **TS2** é disparado. A conexão é liberada após a recepção de uma **DC** ou de uma **DR**, ou com o vencimento do prazo do temporizador TS2.

Ao leitor interessado na especificação informal completa do Protocolo de Transporte Classe 2, sugere-se uma consulta a [CCIT 88b].

6. Metodologia Proposta

Segundo [HoMu 84], o processo de desenvolvimento de um software, enfrenta os seguintes problemas : custo, complexidade, confiabilidade e eficiência.

Segundo [Staa 83], o esforço dispendido no desenvolvimento de software cresce exponencialmente com o aumento da complexidade dos sistemas. O custo de desenvolvimento de software é atualmente dominante num ambiente de computação [WaGu 82]. A facilidade de manutenção e a transportabilidade do código podem permitir a redução de parte desse custo.

A confiabilidade do software depende tanto da sua correção, quanto da robustez para o tratamento de anomalias. Em [BaPe 84], é constatado que grande parte dos erros de programação decorrem de falhas na especificação, seja por erro de especificação, seja por erro de interpretação.

A correção pode ser garantida mediante um processo de desenvolvimento bem estruturado e com facilidades para verificação de propriedades requeridas ao software. Para tal, esse processo precisa deixar de ser uma atividade artesanal e pessoal.

As primeiras soluções, propostas pela Engenharia de Software, basearam-se no lema "Dividir para Conquistar", onde programas complexos são divididos em módulos mais simples. Mais tarde, foram propostos métodos estruturados para programação e desenvolvimento de sistemas. Entretanto, a implementação de um protocolo de comunicação é mais complexa, pois requer que duas implementações independentes se comuniquem corretamente.

Os primeiros protocolos foram especificados informalmente, permitindo interpretações individuais para o mesmo objeto

descrito e gerando, conseqüentemente, implementações incompatíveis. A utilização de métodos formais na especificação de protocolos elimina as ambigüidades da especificação informal.

A especificação formal permite também a verificação de diversas propriedades e a geração automática de códigos. A geração automática de códigos a partir de especificações formais de protocolos foi inicialmente proposta em [NBS 81] e [PoSm 82].

Os primeiros trabalhos sobre implementação semi-automática de protocolos com a linguagem Estelle foram realizados por Bockmann [BGS 84, BGS 86 e BGS 87] na Universidade de Montreal (Canadá). No Brasil, Souza [Souz 90] utilizou o compilador Estelle, construído por Ferneda [Fern 88], para validar protocolos por simulação.

Neste trabalho, queremos propor uma metodologia para implementação semi-automática de protocolos de comunicação, a partir de uma especificação em Estelle. A metodologia proposta pode ser resumida nas seguintes etapas:

(a) obtenção da especificação formal:

- obtenção da especificação formal genérica a partir da especificação informal;
- detalhamento da especificação formal;
- validação da especificação através de simulação;
- validação do design do protocolo através de simulação;

(b) geração do código da implementação:

- adição dos módulos de interface;
- geração do código de implementação;
- implementação manual do código dependente do ambiente operacional (funções de interface e adaptação das rotinas de suporte);
- compilação do código da implementação.

Os principais objetivos dessa metodologia são:

- reduzir a complexidade da implementação do protocolo, decompondo-o em vários módulos;
- retirar as ambigüidades da especificação informal, com a utilização de uma TDF para a especificação do protocolo;
- produzir especificações fáceis de serem adaptadas a escolhas particulares de implementação. Durante o detalhamento da especificação, as decisões particulares são colocadas em funções e procedimentos. A forma de implementação dessas subrotinas é uma escolha particular;
- validar a especificação, através de simulação, antes da geração do código, garantindo a correção do protocolo;
- gerar automaticamente grande parte do código da implementação, garantindo a conformidade entre a especificação e a implementação. O código gerado automaticamente é impessoal e de fácil manutenção. A geração automática permite também aumentar a produtividade da programação e reduzir os erros;
- facilitar a adaptação do código da implementação a qualquer ambiente operacional. O código produzido é transportável para qualquer sistema operacional, pois uma

biblioteca de funções realiza a interface entre a implementação e o ambiente. Essa interface implementa a comunicação com as camadas adjacentes e realiza a adaptação específica das rotinas de suporte (obtenção de informações de temporização, comunicação entre processos do sistema operacional, etc). Cada adaptação específica resume-se na implementação específica dessa interface.

Uma vez que certas escolhas particulares são realizadas na especificação, atualizações da especificação gerarão diferentes implementações.

6.1. Especificação Formal Abstrata

Nesta etapa, uma especificação formal em Estelle é derivada, a partir das recomendações produzidas por órgãos internacionais de padronização. Essa especificação formal é abstrata, na medida em que os detalhes relativos à implementação são omitidos.

Tal especificação deve apresentar as seguintes características:

- os detalhes das estruturas de dados são omitidos, ou seja, permanecem indefinidos;
- decisões de implementação são deixadas a cargo de subrotinas (funções e procedimentos);
- os algoritmos das subrotinas podem ser explicitados mas não implementados;
- as ações das transições são descritas genericamente.

6.1.1. Especificação Abstrata do TP2

A arquitetura Estelle, para a especificação abstrata do TP2, é apresentada na Figura 6.1. O módulo **TP2**, que representa uma entidade de transporte, é refinado em N instâncias do módulo "Abstract Transport Protocol (**ATP**)" e 01 instância do módulo **MAPPING.**

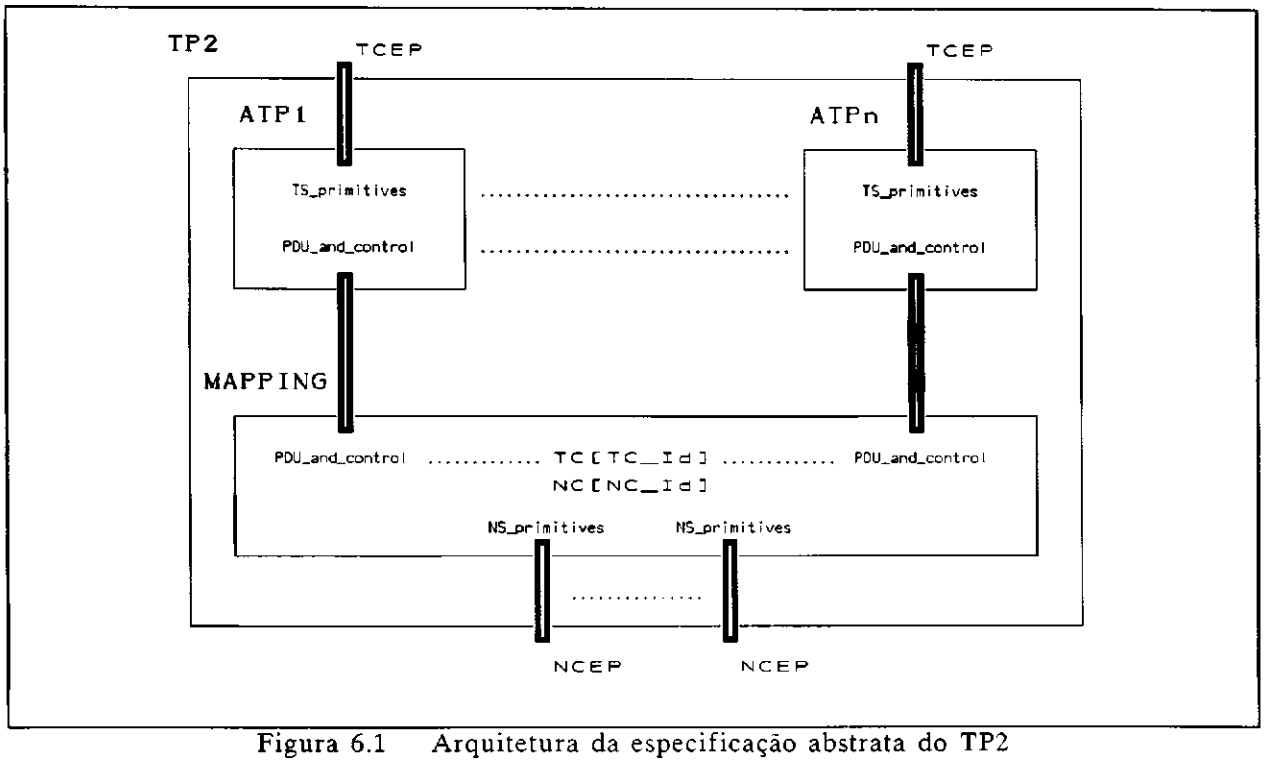

Figura 6.1 Arquitetura da especificação abstrata do TP2

No módulo **ATP** está embutida a lógica do protocolo TP2. O seu comportamento deve refletir as diferentes fases de uma conexão de transporte. Cada conexão de transporte possui sua própria instância **ATP.**

O módulo **MAPPING** é responsável pelo estabelecimento e encerramento das conexões com a rede, pela associação dessas conexões às conexões de transporte (realizando a multiplexação se for necessário) e pelo mapeamento das TPDUs para o parâmetro **dados do usuário** das SPs do tipo **N_DATA.** Uma única instância **MAPPING** gerencia todas as conexões de transporte da entidade **TP2.**

No canal **TS_primitives** (Figura 6.2) são especificadas as SPs trocadas entre as entidades de transporte e seus usuários. Ao desempenhar o papel **T_user**, um usuário pode emitir as SPs T_CONNECT_req, T_CONNECT_resp, T_DISCONNECT_req, T_DATA_req, T_EXPEDITED_DATA_req e deve receber as SPs T_CONNECT_ind, T_CONNECT_conf,T_DISCONNECT_ind,T_DATA_ind, T_EXPEDITED_DATA_ind. O inverso ocorre com uma entidade de transporte, que desempenhará o papel **T_provider**.

```
channel TS_primitives (T_user, T_provider);
  by T_user :
     T_CONNECT_req    (Called_Address, Calling_Address: TSAP_address_type;
                       Expedited_Data_Option: boolean;
                       Quality_of_Service: QOS_type;
                       TS_User_data: optional_TSDU);
     T_CONNECT_resp   (Responding_Address: TSAP_Address_type;
                       Expedited_Data_Option: boolean;
                       Quality_of_Service: QOS_type;
                       TS_User_data: optional_TSDU);
     T_DISCONNECT_req(TS_User_data: optional_TSDU);
     T_DATA_req       (TS_user_data: TSDU);
     T_EXPEDITED_DATA_req (TS_user_data: TSDU);
  by T_provider:
     T_CONNECT_ind  (Called_Address, Calling_Address: TSAP_address_type;
                     Expedited_Data_Option: boolean;
                     Quality_of_Service : QOS_type;
                     TS_User_data : optional_TSDU);
     T_CONNECT_conf (Responding_Address: TSAP_Address_type;
                     Expedited_Data_Option: boolean;
                     Quality_of_Service : QOS_type;
                     TS_User_data : optional_TSDU);
     T_DISCONNECT_ind (TS_user_data : optional_TSDU;
                       Reason : TS_disconnect_reason_type);
     T_DATA_ind       (TS_user_data : TSDU);
     T_EXPEDITED_DATA_ind (TS_user_data : TSDU);
```

Figura 6.2 Canal TS_primitives

No canal **NS_primitives** (Figura 6.3), são especificadas as SPs trocadas entre as entidades e a camada de rede. Ao desempenhar o papel **N_user**, uma entidade de transporte pode enviar as SPs N_CONNECT_req, N_CONNECT_resp, N_DISCONNECT_req, N_DATA_req, N_DATA_ACKNOWLEDGE_req, N_EXPEDITED_DATA_req, N_RESET_req, N_RESET_resp e deve receber N_CONNECT_ind, N_CONNECT_conf, N_DISCONNECT_ind, N_DATA_ACKNOWLEDGE_ind,

N_DATA_ind, N_EXPEDITED_DATA_ind, N_RESET_ind, N_RESET_conf. O inverso ocorre com a camada de rede, que desempenhará o papel **N_provider.**

```
channel NS_primitives (N_user, N_provider);
  by N_user :
     N_CONNECT_req   (Called_Address, Calling_Address: NSAP_address_type;
                      Receipt_Confirm_Option: boolean;
                      Expedited_Data_Option: boolean;
                      Quality_of_Service: NQOS_type;
                      NS_User_data: optional_NSDU);
     N_CONNECT_resp  (Responding_Address: NSAP_Address_type;
                      Receipt_Confirm_Option: boolean;
                      Expedited_Data_Option: boolean;
                      Quality_of_Service: NQOS_type;
                      NS_User_data: optional_NSDU);
     N_DISCONNECT_req (Reason: NS_disconnect_reason_type;
                       NS_User_data: optional_NSDU;
                       Responding_Address: NSAP_Address_type);
     N_DATA_req (NS_user_data: NSDU; Confirmation_Req: boolean);
     N_DATA_ACKNOWLEDGE_req; (* not used in class 0 and 2 *)
     N_EXPEDITED_DATA_req (NS_user_data: NSDU);
     N_Reset_Req (Reason: NS_reset_reason_type);
     N_Reset_Resp;
  by N_provider :
     N_CONNECT_ind   (Called_Address, Calling_Address: NSAP_address_type;
                      Receipt_Confirm_Option: boolean;
                      Expedited_Data_Option: boolean;
                      Quality_of_Service: NQOS_type;
                      NS_User_data: optional_NSDU);
     N_CONNECT_conf  (Responding_Address: NSAP_Address_type;
                      Receipt_Confirm_Option: boolean;
                      Expedited_Data_Option: boolean;
                      Quality_of_Service: NQOS_type;
                      NS_User_data: optional_NSDU);
     N_DISCONNECT_ind (Originator: NS_Originator_type;
                       Reason : NS_disconnect_reason_type;
                       NS_User_data:optional_NSDU;
                       Responding_Address:NSAP_Address_type);
     N_DATA_ind (NS_user_data: NSDU; Confirmation_Req: boolean);
     N_DATA_ACKNOWLEDGE_ind;
     N_EXPEDITED_DATA_ind (NS_user_data: NSDU);
     N_Reset_Ind (Reason: NS_reset_reason_type;
                  Originator: NS_Originator_type);
     N_Reset_Conf ;
```

Figura 6.3 Canal NS_primitives

Na Figura 6.4 é apresentado um fragmento da especificação abstrata do TP2, onde são ilustrados os mecanismos de Estelle que permitem a omissão dos detalhes irrelevantes para essa fase.

```
01 Specification TP2;
02 type
03   Octet = ...;
04   String_of_octets = ...;
05   TSAP_address_type = ... ;
06   QOS_type= ...;
07   ***
08 channel TS_primitives (T_user, T_provider);
09  ***
10 channel NS_primitives (N_user, N_provider);
11  ***
12 module ATP_ENTITY process;
13  ip TSAP  : TS_primitives   (T_Provider) individual queue;
14     MAPX  : PDU_and_control (protocol)   individual queue;
15 end (*ATP_ENTITY*);
16 body ATP_ENTITY_BODY for ATP_ENTITY;
17   var
18     local_TPDU : TPDU_and_control_information;
19   initialize to CLOSED
20   ***
21   function definitive_Class(Class, Alt_Class:class_type;
22                             QOS:QOS_type): class_type;
23   primitive;
24   {se alguma das classes propostas estiverem implementadas
25    e a qualidade de serviço puder ser atendida, selecionar
26    a classe preferencial ou então as alternativas}
27   ***
28   trans
29     when TSAP.T_CONNECT_resp
30     provided Proposition_Accepted( *** )
31     from OPEN_IN_PROGRESS_CALLED to OPEN
32     begin
33       class:=definitive_class(class, alt_Class, QOS);
34       with local_TPDU do
35         begin
36           kind := CC;
37           class_ind:=class;
38           ***
39         end;
40       output MAPX.dados (local_TPDU);
41     end;
42   ***
43 end;
```

Figura 6.4 Fragmento da Especificação Formal Abstrata

Em relação à especificação apresentada na Figura 6.4, os detalhes das estruturas de dados (linhas 1 a 2) são omitidos utilizando-se a construção ... . O algoritmo da função **definitive_Class** (linhas 4 a 7) é explicitado, através de comentários, e a palavra reservada **primitive** indica que este será posteriormente implementado. A ação da transição (linhas 13 a 18) é descrita genericamente.

As transições do módulo ATP descrevem as seguintes fases do protocolo TP2 (Figura 6.5): estabelecimento de conexão (linhas 1 a 10), transmissão de dados e controle de fluxo (linhas 12 a 28) e encerramento de conexões (linhas 30 a 51).

As principais subrotinas (funções e procedimentos) internas ao ATP podem ser agrupados em: gerência de buffers, rotinas auxiliares para controle de fluxo, rotinas que implementam a negociação de opções e de classe de protocolo e rotinas auxiliares genéricas.

O ATP possui um módulo interno T_CLOCK que representa os temporizadores (opcionais). Um canal interno T_Clock_Access (Figura 6.6) é definido para ligar os módulos ATP e T_CLOCK. As primitivas **T_SET_CLOCK** e **T_STOP_CLOCK** permitem ao ATP (no papel **User**) ligar e desligar o temporizador, respectivamente. Ocorrendo o fim da temporização, T_CLOCK (no papel **Provider**) envia para o ATP uma primitiva **T_TIME_OUT**.

```
01 (* estabelecimento de conexao *)
02 trans
03    when TSAP.T_CONNECT_req
04    provided not In_Use
05    from CLOSED
06    to   OPEN_IN_PROGRESS_CALLING
07    begin
08      ***
09      output MAPX.dados (local_TPDU);
10    end;
11 ***
12 (* encerramento de conexão *)
13 trans
14    when TSAP.T_DISCONNECT_req
15    provided (class = class_2)
16    from OPEN to CLOSING
17    begin
18      kind := DR;
19      output MAPX.dados (local_TPDU);
20    end
21
22 ***
23    when MAPX.dados(TPDU)
24    provided (TPDU.kind = DC)
25    from CLOSING to CLOSED
26    begin
27      output MAPX.terminated;
28    end;
29 ***
30 (* transmissão de dados *)
31 trans
32   when TSAP.T_DATA_req (* TS_user_data *)
33   provided (Class=Class_2) and
34             (enough_space(Send_buffer,T_Data_UD_length))
35   from OPEN
36   to   OPEN
37   begin
38     append(Send_buffer, TS_user_data,true);
39   end;
40 ***
41 trans
42   provided (((S_credit <> 0)  and not(no_flow_cont)) or
43   (no_flow_cont)) and  (not_empty(send_buffer))
44   begin
45     with local_TPDU do
46       begin
47         kind:= DT;
48         get_next_fragment(send_buffer,DT_length,user_data);
49       end;
50     output MAPX.dados(local_TPDU);
51   end;
```

Figura 6.5 Exemplos de Transições do TP2

```
channel T_Clock_Access (User,Provider);
  by User     :
     T_Set_Clock (delay_time:time_type);
     T_Stop_Clock;
  by Provider :
     T_Time_Out;
```

Figura 6.6 Canal T_CLOCK_ACCESS

Funcionalmente, as transições do módulo MAPPING podem ser agrupadas em: gerência das conexões de rede (utilizando as SPs N_CONNECT, N_DISCONNECT, N_RESET), transmissão e recepção de dados (utilizando a SP N_DATA), associação das TCs às NCs, associação das TPDUs às TCs. As principais subrotinas do módulo MAPPING podem ser agrupadas em: rotinas associadas à função de codificação e decodificação das TPDUs, rotinas de gerência de conexões, rotinas de mapeamento e gerência dos parâmetros das SPs trocadas entre as camadas adjacentes.

Em relação ao tratamento de erros gerados pelo usuário ou pela entidade par, essa tarefa foi dividida entre os módulos ATP e Mapping.

O módulo ATP é responsável pelo tratamento de erros que geram exceções na tabela de transições do TP2 e outros erros de controle, tais como :

- TPDUs recebidas fora de seqüência;
- problemas relacionados ao controle de fluxo e numeração das TPDUs **DT**;
- serviços incompatíveis para as opções e a classe selecionadas. Por exemplo, tentativa de utilizar o serviço de troca de dados expressos, quando essa opção não estiver selecionada ou quando a classe selecionada for classe 0.

O ATP controla também a temporização associada ao envio da TPDU **CR** (temporizador TS1) e da TPDU **DR** (temporizador TS2).

O módulo MAPPING trata erros relacionados à codificação e à decodificação de TPDUs, tais como:

- parâmetro inválido ou valor do parâmetro inválido para o tipo de TPDU;
- parâmetro inválido ou valor do parâmetro inválido para a classe de serviço negociada;
- erro na decodificação dos bits.

Também é tratada como erro a impossibilidade de alocação ou manutenção de uma NC, que atenda aos requisitos da TC.

A comunicação entre os módulos ATP e MAPPING é realizada por um canal do tipo **PDU_and_control** (Figura 6.7). A primitiva de interação **dados** contém todos os dados necessários para compor uma TPDU e pode fluir nos dois sentidos do canal. O ATP envia para o MAPPING as primitivas **terminated** (encerramento explícito da TC), **implicit_terminated** (encerramento implícito da TC na classe 0) e **Reset_Net** (solicitação de reinicialização da NC). O MAPPING envia para o ATP as primitivas **protocol_error_indication** (indicação de erro de protocolo detectado por MAPPING) e **close_indication** (indicação de fechamento da NC).

```
channel  PDU_and_control (protocol, mapping);
 by protocol,mapping :
    dados (TPDU :TPDU_and_control_information);
 by protocol:
    terminated;
    implicit_termination;
    Reset_Net;
 by mapping:
    protocol_error_indication
                    (reason: TS_disconnect_reason_type);
    close_indication (reason : TS_disconnect_reason_type);
```

Figura 6.7 Canal PDU_and_control

6.2. Especificação Formal Detalhada

Nesta etapa, as lacunas deixadas na especificação abstrata devem ser preenchidas, buscando-se uma especificação orientada para implementação, mas ainda independente do ambiente operacional. Tal especificação deve apresentar as seguintes características:

- detalhes das estruturas de dados são descritos;
- os algoritmos das subrotinas são implementados;
- as ações das transições são descritas suscintamente;
- novas transições e subrotinas são adicionadas, procurando tornar o protocolo mais robusto em relação ao tratamento das exceções;
- as transições espontâneas são reduzidas, a fim de melhorar a eficiência da implementação.

6.2.1. Especificação Detalhada do TP2

As escolhas particulares vão refletir na definição das estruturas de dados e no comportamento do protocolo.

Os principais detalhes das estruturas de dados do TP2, que devem ser descritos, são:

- o formato do endereço de rede segue a recomendação X.121 (40 dígitos) [CCIT 88f];

- o endereço de transporte (Figura 6.8) é formado pela associação dos endereços de rede (40 dígitos) e sessão (16 octetos);

- embora o protocolo possa operar tanto nas classes 0 e 2, a classe 2 é a classe de protocolo preferencial;

- os dois formatos de TPDU (normal e estendido) são suportados pelo protocolo, mas o formato normal é sugerido como default;

- o protocolo dá suporte a todos os tamanhos de TPDU possíveis, mas um tamanho máximo foi escolhido (8192 octetos);

- uma das opções de serviço escolhida foi a utilização do controle de fluxo explícito;

- foram determinadas as características (definidas pelo usuário) do campo "protection" das TPDUs **CR** e **CC**;

- foram fixadas as características do campo "throughput" das TPDUs **CR** e **CC** (24 octetos, ou seja, a parte opcional também foi utilizada);

- foi definido que os valores de temporização são obtidos do parâmetro "qualidade de serviço". Atraso no estabelecimento ("establishment delay") para TS1

(temporizador disparado após envio de uma TPDU **CR**) e atraso no encerramento ("release_delay") para TS2 (temporizador disparado após envio de uma TPDU **DR**);

- foram definidas as características da estrutura de dados "string_of_octets". O campo "length" armazena o tamanho e o campo "contents" armazena o conteúdo propriamente dito;

- o tipo "Octet" foi implementado como tipo "char";

- os tipos "seq_number_type" e "credity_type" foram implementados como "Long_Int" (variando de 0 a 2147483647), dando-se suporte ao formato estendido;

- o tipo "time_type", utilizado para definir variáveis de temporização, é do tipo "integer";

- alguns campos das TPDUs foram implementados como "array [**n**] of octet", onde **n** é a quantidade de octetos definida na recomendação.

```
NSAP_address_type = array [1..40] of Digit; (16 ISP + 24 DSP)
SSAP_address_type = array [1..16] of Octet;
TSAP_address_type =
   record
     NETWORK : NSAP_address_type;
     SESSION : SSAP_address_type;
   end;
```

Figura 6.8 Estrutura do endereço de Transporte

Com relação às subrotinas e às transições, foram adotadas as seguintes estratégias:

- dois temporizadores (opcionais, segundo [CCIT 88b]) são utilizados no auxílio das funções de estabelecimento e encerramento de conexões;

- em relação ao controle de fluxo, os créditos são concedidos de acordo com a disponibilidade de "buffer" de recepção;

- o "buffer" foi implementado da forma mais simples, ou seja, utilizou-se alocação estática (e predefinida) de blocos de tamanho fixo (Figura 6.9);

- foram acrescentadas diversas transições e subrotinas para tratamento de erros do protocolo;

- o mecanismo para mapear o endereço de rede no endereço de transporte foi facilitado pelo formato do endereço de transporte;

- foram definidos, sumariamente, os mecanismos para mapeamento das mensagens de erro de uma SP da camada de rede para uma SP da camada de transporte, a forma para traduzir os parâmetros de qualidade de serviço das SPs N_CONNECT para T_CONNECT, a forma de negociação de valores de parâmetros (utilização dos mecanismos de controle de fluxo explícito, comprimento e formato de TPDU, classe de protocolo);

- a associação entre as TCs e as NCs é feita com o auxílio das tabelas de conexões de rede e transporte (Figura 6.10), considerando-se a qualidade de serviço e a classe de protocolo.

```
data_buffer =
  record
    Next_Block : Integer; (* Tail *)
    First_Block: Integer; (* Head *)
    space      : array [1..Nr_Buffer_block] of integer;
    Data       : array [1..Max_Buffer_Size] of octet;
  end;
```

Figura 6.9 Estrutura do "buffer"

```
TC_table  =
   record
     in_use           : boolean;
     local_T_addr,
     remote_T_addr    : TSAP_address_type ; (* see TS *)
     local_ref,
     remote_ref       : reference_type;
     assigned_NC      : NCEP_Id_Type;
     max_PDU_size     : integer;
     class            : class_type;
     Options          : Option_type;
     ED_Option        : boolean;
     QTS              : QOS_type; (* see TS *)
     PDU_buffer_full: array [TPDU_code_type] of boolean;
     PDU              : TPDU_and_control_information
   end;

NC_table =
   record
     NC_state         : (CLOSED, OPEN_IN_PROGRESS,
                         WAIT_BEFORE_CLOSING, OPEN);
     remote_N_addr ,
     local_N_addr     : NSAP_address_type; (* see NS *)
     this_side        : both_sides ; (* see NS *)
     QNS              : NQOS_type;   (* QOS da NC *)
     QNS_used         : NQOS_type;   (* QOS das TCs assoc *)
     received_NSDU,
     NSDU_to_be_sent:
       record
         user_data_present : boolean;
         data : string_of_octets;
       end;
     no_multiplexing      : boolean;
     corresponding_TC_id : TCEP_id_type;
     N_PDU_buffer_full: array [TPDU_code_type] of boolean;
     N_PDU              : TPDU_and_control_information;
   end;
```

Figura 6.10 Estrutura da Tabela de conexões

As situações de exceção podem ser tratadas como erro ou ignoradas. Para isso, algumas transições foram adicionadas:

- em caso de "time-out" de qualquer temporizador, a conexão deve ser liberada (Figura 6.11);

- na ocorrência de erros de protocolo, uma TPDU **ER** deve ser enviada e o temporizador TS2 deve ser ativado (Figura 6.12). Por exemplo, recepção de uma TPDU ou de uma

SP inesperada (e.g., a TPDU **ED** não pode ser recebida se a opção "dados expressos" não estiver selecionada ou se a classe de protocolo for a classe zero ou se a conexão estiver fechada), seqüência de numeração errada, recepção de dados fora da janela de recepção (devido a inexistência de créditos);

- a recepção de TPDUs que não podem ser associadas a uma TC gera uma mensagem de desconexão ao emissor (TPDU DR);

- o protocolo procura determinar uma desconexão anormal sinalizada e monitorar o envio descontrolado de dados expressos pela entidade par;

- na fase de estabelecimento de conexão, o protocolo verifica a existência de conexões de rede disponíveis.

```
When Timer2.T_Time_Out
provided Protocol_Error_State and (Class=Class_2)
to CLOSING
begin
   (* protocol error *)
   output TSAP.T_Disconnect_Ind(NULL_DATA,133);
   with local_TPDU do
     begin
       kind:=DR;
       disconnect_reason:=133;
       User_Data:=NULL_DATA;
     end;
   output Timer2.T_Stop_Clock; (* ER *)
   output Timer2.T_Set_Clock(QTS_Ind_s.Release_Delay);
   output MAPX.dados(local_TPDU);
end;
```

Figura 6.11 Ação resultante do estouro do Temporizador

```
01   (*- TPDUs inválidos para classe_0  -*)
02   when MAPX.dados(TPDU)
03   provided ((TPDU.kind = DR) or (TPDU.kind = AK) or
04             (TPDU.kind = EAK) or (TPDU.kind = EDT) or
05             (TPDU.kind = DC)) and (class=Class_0)
06  from OPEN to OPEN
07   begin
08     protocol_error(local_TPDU,invalid_TPDU);
09     output MAPX.dados(local_TPDU);
10     output Timer2.T_Set_Clock(QTS_ind_s.Release_Delay);
11   end;
```

Figura 6.12 Trecho para tratamento de erros

As subrotinas destinadas à codificação e à decodificação de TPDUs devem também detectar parâmetros errados e atribuir TPDUs às respectivas TCs. Diversas rotinas auxiliares de conversão de tipo e de manipulação da tabela de conexões (TC_table e NC_table) foram também codificadas.

As funções desempenhadas pelas subrotinas do ATP podem ser agrupadas da seguinte forma:

(a) gerência de "buffers"

- init_buffer, que inicializa o "buffer";
- enough_buffer, que verifica o espaço disponível no "buffer;
- append, que adiciona dados ao "buffer";
- not_empty, que verifica se existe dados no "buffer";
- is_end_of_TSDU, que indica a leitura de uma TSDU completa;
- get_next_fragment, que obtém um bloco do "buffer";

(b) rotinas auxiliares para controle de fluxo

- Next_Seq, que obtém o próximo nº de seqüência;

- Credit_Increm, que verifica se podem ser concedidos mais créditos;

- New_Credit: calcula o nº de créditos a ser concedido;

- Subtract_CDT, que subtrai o nº de créditos;

(c) rotinas para a negociação das opções e para a negociação da classe de protocolo

- Proposition_Accepted, que verifica se, durante a negociação dos parâmetros, os valores respondidos são inferiores aos sugeridos;

- Alt_Class_Set, que retorna as classes alternativas implementadas pelo protocolo;

- Implemented_Class, que retorna a classe preferencial implementada pelo protocolo;

- Definitive_Class, que retorna a classe na qual o protocolo irá operar;

(d) rotinas auxiliares genéricas

- Protocol_Error, que monta uma TPDU ER, correspondente ao erro do protocolo;

- Min, que retorna o valor mínimo dentro de uma lista de valores inteiros.

As funções desempenhadas pelas subrotinas do Mapping podem ser agrupadas da seguinte forma:

(a) rotinas associadas à função de codificação e decodificação de TPDUs

- decode_PDU, que obtém uma TPDU do campo "dados do usuário" da SP N_DATA_ind;

- encode_PDU, que armazena uma TPDU no campo "dados do usuário" da SP N_DATA_req;

- put_TSAP_add, que grava o endereço de transporte;

- div_byte, que decompõe uma variável LongInt em 4 bytes;

- get_TSAP_add, que obtém o endereço de transporte;

- limit_User_Data, que limita o campo "dados do usuário" ao tamanho máximo permitido;

- assign_User_Data, que obtém o campo "dados do usuário" de uma TPDU;

- get_TPDU_code, que obtém o código do tipo de TPDU;

- get_Class, que obtém a classe de protocolo;

- Get_Char, que obtém um caractere de uma variável do tipo "string_of_octets";

- Put_char, que grava um caractere numa variável do tipo "string_of_octets";

- CharToBin, que retorna os bits de um dado do tipo "char";

- length_error, que avalia se o tamanho do campo "dados do usuário" está correto;

- ordX, que gera o código ASCII do caractere;

- Concat_String, que concatena variáveis do tipo "string_of_octets";

(b) rotinas de gerência de conexões

- determine_TC, que retorna a conexão de transporte identificada pela referência;

- clear_buffer_full, que reinicializa o "buffer" de uma conexão de transporte;

- find_same_pair, que verifica se existe uma conexão com a mesma referência;

- Get_TC, que obtém uma conexão de transporte;

- Get_NC, que obtém uma conexão de rede;

- Assign_NC, que associa uma conexão de transporte a uma conexão de rede existente;

- comp_Naddr, que compara campos que contém endereços de rede;

- Next_Ref, que obtém o próximo nº de seqüência, a ser utilizado para referenciar uma conexão de transporte;

- clear_NC_buffers, que limpa "buffers" das conexões de rede;

- close_NC, que libera as estruturas de dados utilizadas para gerenciar uma conexão de rede;

no_TC_assigned, que verifica se existe alguma conexão de transporte associada à conexão de rede;

- close_and_clear_buffers, que libera "buffers" e estruturas de dados (de gerência) associadas a uma conexão de transporte;

(c) rotinas de tratamento dos parâmetros das SPs trocadas entre as camadas adjacente

- map_T_reason, que realiza o mapeamento do código de erro gerado na camada de rede para a camada de transporte;

- QTS_in, que verifica se os valores da qualidade de serviço são superiores aos indicados pelo iniciador da conexão;

- enough_space_NSDU, que verifica se existe espaço no campo "dados do usuário" da primitiva N_DATA_req para transportar a TPDU;

- determine_N_address, que obtém o endereço de rede a partir do endereço de transporte;

- QTS_eval, que indica se a qualidade de serviço pode ser atendida;

- class_eval, que indica se a classe é suportada pelo protocolo;

- update_QNS, que atualiza o parâmetro "qualidade de serviços" da camada de rede com a qualidade de serviços da camada de transporte;

- map_QTS, que realiza o mapeamento do parâmetro "qualidade de serviços" da camada de transporte para "qualidade de serviços" da camada de rede;

- QNS_support, que verifica se a "qualidade de serviço" da camada de rede pode atender a "qualidade de serviço" solicitada pelo usuário de transporte.

As subrotinas implementadas na linguagem Pascal foram testados externamente à especificação para garantir a correção do algoritmo.

6.3. Validação da Especificação e do Design do Protocolo

A especificação formal não garante, por si só, que o protocolo não venha a apresentar comportamento anômalo. Dessa forma, é importante que a especificação seja verificada e validada.

A verificação das propriedades de um protocolo especificado em Estelle pode ser feita através da técnica de Análise de Alcançabilidade (MEF) combinada com Prova de Programas (parte da especificação escrita na linguagem Pascal).

Nesta etapa, as validações são realizadas empregando-se a ferramenta **SIMULATOR/DEBUGGER** do ambiente EWS. Para que uma especificação detalhada possa ser simulada é necessário:

- submeter a especificação ao editor EWSEDIT;
- gerar o código intermediário, a partir da especificação detalhada, utilizando **EWSTRANS**;
- gerar o código C orientado para simulação, a partir do código intermediário, utilizando **EWSGEN**. No caso de simulação, o compilador C é automaticamente acionado para gerar o código objeto;
- ligar o código objeto às rotinas de simulação, utilizando o comando **GOEWSMAKESIMU**.

Durante a simulação, o usuário seleciona uma transição para ser disparada. É possível analisar o valor das variáveis (variáveis internas, variáveis exportadas e variáveis de estado) e o conteúdo dos parâmetros das primitivas, após a execução de cada transição. Seqüências de transições podem ser executadas, armazenadas e reexecutadas.

6.3.1. Validação da Especificação do TP2

A validação do TP2 foi realizada em dois passos: simulação parcial, onde somente o módulo **ATP** é analisado, e simulação do protocolo TP2, onde os módulos **ATP** e **MAPPING**, que compõe a entidade de transporte, são analisados.

Para a simulação parcial foi utilizada a arquitetura apresentada na Figura 6.13. Cada instância **T_USER** desempenha o papel de uma entidade de sessão e está conectada a uma instância **ATP**. As duas instâncias **ATP** estão por sua vez conectadas ao módulo **MAP_NET**, que desempenha os papéis do módulo **MAPPING** e da camada rede.

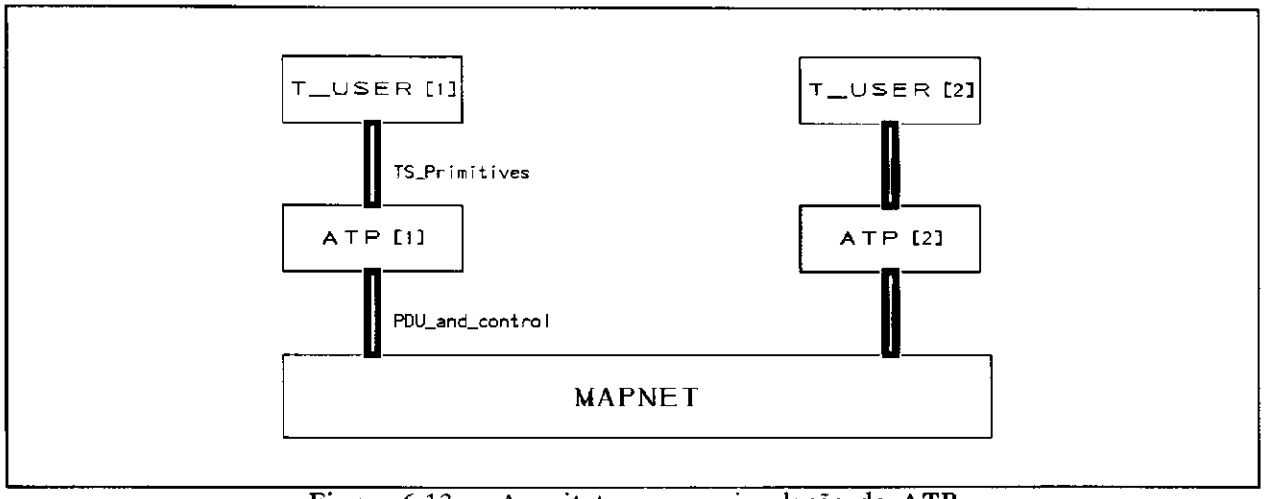

Figura 6.13 Arquitetura para simulação do ATP

A simulação parcial tem por objetivo a validação da MEF associada ao TP2. Neste sentido, é aplicado ao módulo **ATP** um conjunto de cenários de teste. Tais cenários possibilitam a análise das seguinte funções do **ATP:**

- abertura de conexões, transmissão de dados e encerramento de conexões;
- tratamento de interações inválidas (SP ou TPDU);
- gerência do controle de fluxo.

Para a simulação do protocolo TP2 foi utilizada a arquitetura apresentada na Figura 6.14. Duas instâncias **TP2** são conectadas ao módulo **Network_Service**, que fornece o serviço da camada rede. Cada instância **TP2** local está conectada a um conjunto de instâncias **T_USER**, através de instâncias **ATP** que, por sua vez, são conectadas a uma instância **MAPPING**.

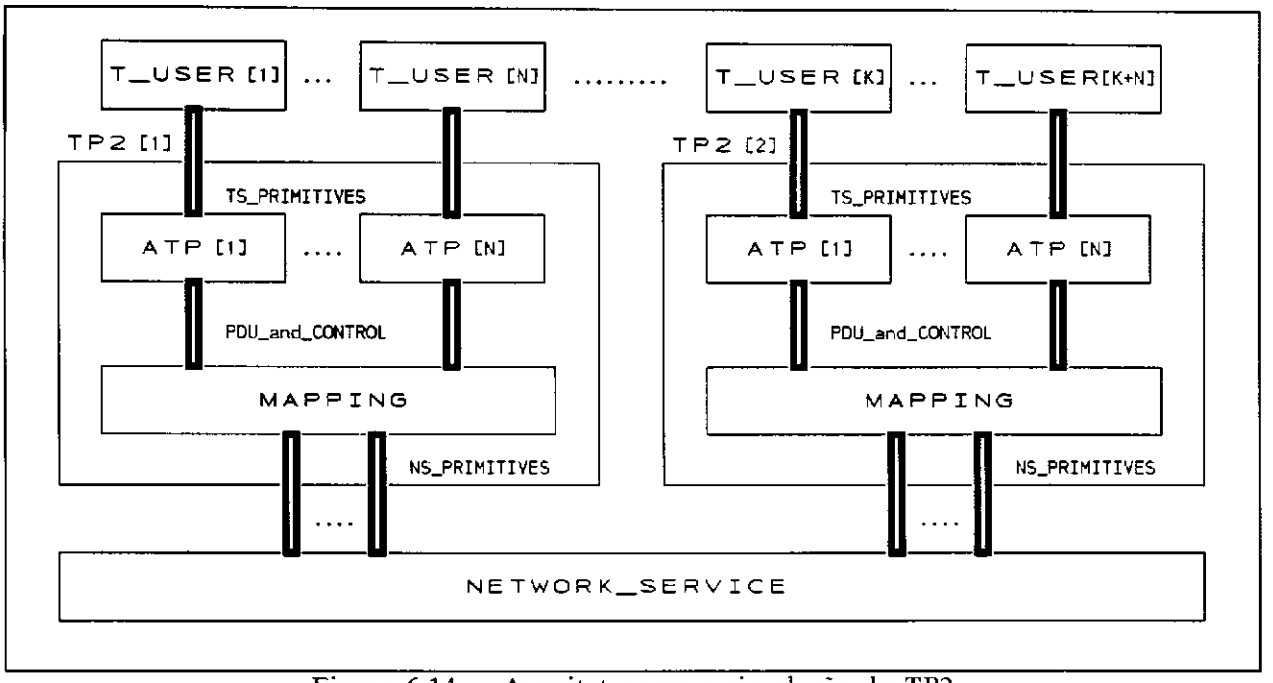

Figura 6.14 Arquitetura para simulação do TP2

Além das funções avaliadas na simulação parcial, os cenários de teste para a simulação do protocolo TP2 possibilitam também a análise das seguintes funções:

- associação das conexões de transporte às conexões de rede;
- multiplexação;
- utilização dos serviços da camada rede, através das SPs que permitem a abertura de conexões (**N_CONNECT**), a transmissão de dados (**N_DATA**) e o encerramento de conexões (**N_DISCONNECT**);
- detecção de erros do protocolo associados à codificação/decodificação de TPDUs (e.g, parâmetros inválidos ou com valores inválidos).

6.3.2. Validação do Design do TP2

A especificação informal do serviço fornecido pela camada de transporte, descrita em [CCIT 88a], pode ser formalizada e

simulada a fim de se obter um conjunto de traços de serviço. A validação do design do protocolo pode então ser realizada, através da comparação desses traços com os traços obtidos quando da simulação do protocolo. Todo este procedimento está detalhado em [Souz 90].

Para a simulação do serviço do TP2, visando a validação do design do TP2, foi utilizada a arquitetura apresentada na Figura 6.15. Cada instância **T_USER** desempenha o papel de uma entidade de sessão e está conectada ao módulo **Transport_Service**, que fornece o serviço de transporte.

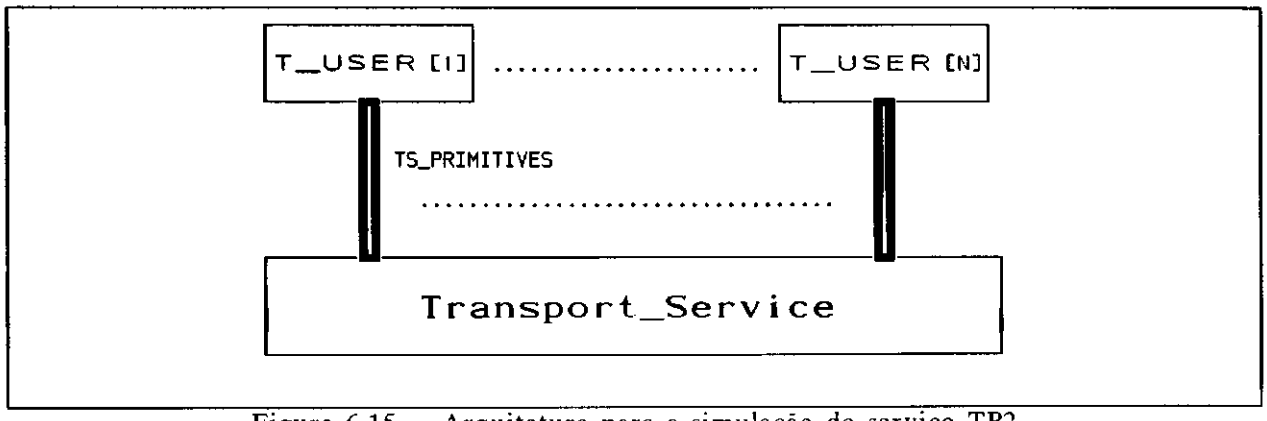

Figura 6.15 Arquitetura para a simulação do serviço TP2

Na simulação do serviço, são retidos, para uma análise posterior, os traços referentes às diversas fases de uma TC. Cada traço é constituído de uma seqüência de SPs, acompanhadas de seus respectivos parâmetros. Para o estabelecimento bem sucedido de uma TC, o seguinte traço foi obtido:

(a) o módulo T_USER[1] emite uma SP **T_Connect_request** ao módulo TRANSPORT_SERVICE;

(b) o módulo T_USER[2] recebe uma SP **T_Connect_indication** do módulo TRANSPORT_SERVICE;

(c) o módulo T_USER[2] envia uma SP **T_Connect_response** ao módulo TRANSPORT_SERVICE;

(d) o módulo T_USER[1] recebe uma SP **T_Connect_confirm** do módulo TRANSPORT_SERVICE.

Para a simulação do protocolo TP2, visando a validação do seu design, foi utilizada a arquitetura apresentada na Figura 6.16.

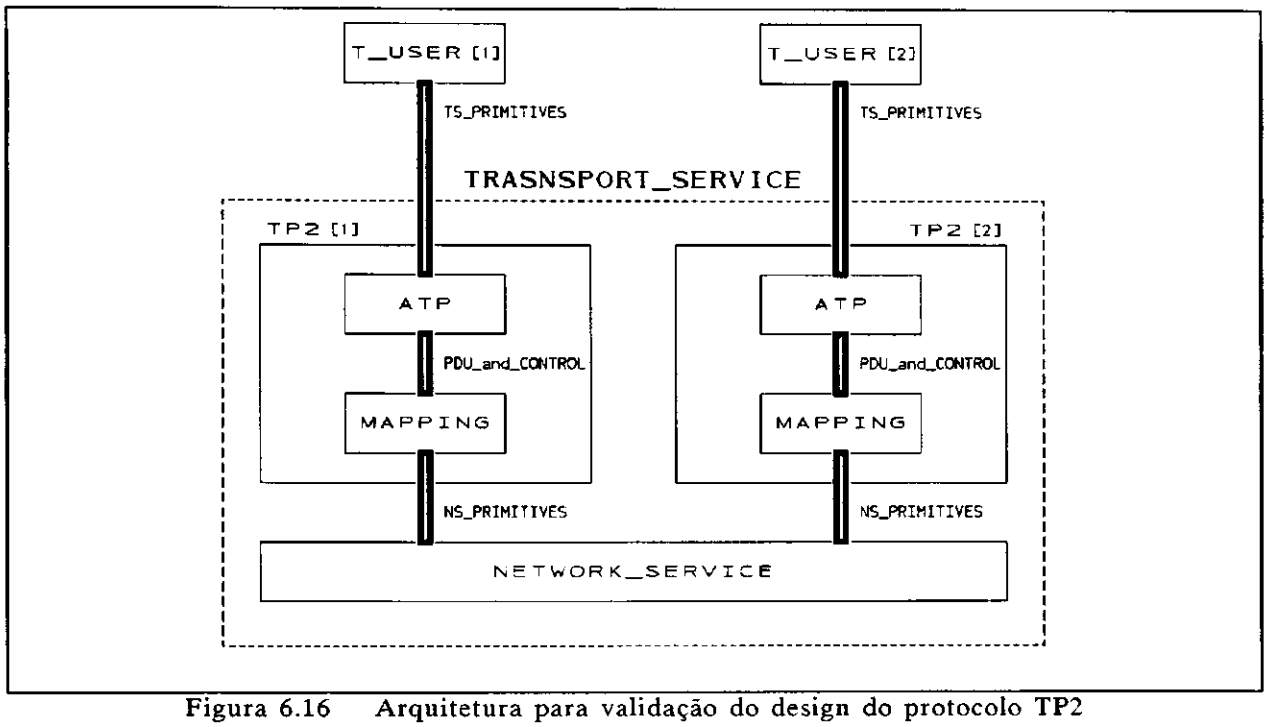

Figura 6.16 Arquitetura para validação do design do protocolo TP2

Nessa simulação, também são retidos, para uma análise posterior, os traços referentes às diversas fases de uma TC. Cada traço é constituído de uma seqüência de SPs e TPDUs, acompanhadas de seus respectivos parâmetros. Para o estabelecimento bem sucedido de uma TC, o seguinte traço foi obtido:

(a') o módulo T_USER[1] emite uma primitiva **T_Connect_request** (Tabela 6.1) ao módulo TRANSPORT_SERVICE:

- na instância TP2[1], essa SP é recebida pelo módulo ATP que, por sua vez, monta uma TPDU **CR** (Tabela 6.2) a partir dos parâmetros da SP T_Connect_request recebida. Essa TPDU é enviada ao módulo MAPPING;

Parâmetro	Conteúdo
Endereço Chamador	00000000000000000000000000000000000001 0000000000000001
Endereço Chamado	00000000000000000000000000000000000002 0000000000000002
Qualidade de Serv.	
- estab_delay	00
- estab_failure	000000
- throughput	001001001001001001001001
- transit_delay	0101010101
- resid_err_rate	111
- Resiliense	000000
- Transf_failure	000000
- Release_delay	00
- Relea_failure	000000
- priority_QOS	0101
- protection	1
Opção D. Expressos	1
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.1 Parâmetros da SP T_Connect_request

Parâmetro	Conteúdo
LI	$(84)_{10}$
CR + CDT	$(11100011)_2$
DST_REF	00
SRC_REF	01
CLASS + OPT	$(00100010)_2$
(parte variável)	
TSAP Chamador	0000000000000001
TSAP Chamado	0000000000000002
TPDU_size	$(00001101)_2$
Version	1
Protection	1
Additional_option	$(00000001)_2$
Alter_Classes	$(00000000)_2$
Qualidade de Serv.	
- throughput	001001001001001001001001
- resid_err_rate	111
- priority_QOS	0101
- transit_delay	0101010101
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.2 Parâmetros da TPDU CR

- no módulo MAPPING da instância TP2[1] tenta-se alocar uma NC já estabelecida. Caso não seja possível, é emitida uma SP **N_Connect_request** ao módulo NETWORK_SERVICE para estabelecer uma NC. Os parâmetros qualidade de serviço e endereço da SP T_Connect_request são mapeados para os respectivos parâmetros da SP N_Connect_request;

- se o provedor da NC puder atender ao pedido, o módulo NETWORK_SERVICE envia uma SP N_CONNECT_indication ao módulo MAPPING da instância TP2[2];

- o módulo MAPPING da instância TP2[2] responde, enviando uma SP N_Connect_response ao módulo NETWORK-SERVICE;

- o módulo NETWORK-SERVICE envia uma SP N_CONNECT_confirm ao módulo MAPPING da instância TP2[1], estabelecendo a **NC**;

- uma vez que foi encontrada uma NC, o módulo MAPPING da instância TP2[1] emite ao módulo NETWORK_SERVICE uma SP **N_Data_request** com uma TPDU **CR** empacotada no campo dados do usuário;

- o módulo NETWORK_SERVICE envia uma SP **N_DATA_indication** à instância TP2[2];

- na instância TP2[2], o módulo MAPPING decodifica a SP N_Data_indication e emite uma TPDU **CR** ao módulo ATP;

- o módulo ATP da instância TP2[2] emite uma SP T_Connect_indication ao módulo T_USER[2]. Os parâmetros da SP T_Connect_indication (endereços, qualidade de serviço, opção para dados expressos, dados do usuário) são montados a partir dos parâmetros da TPDU CR;

(b') módulo T_USER[2] recebe uma SP **T_Connect_indication** (Tabela 6.3) do módulo TRANSPORT_SERVICE;

Parâmetro	Conteúdo
Endereço Chamador	00000000000000000000000000000000000000000001 0000000000000001
Endereço Chamado	00000000000000000000000000000000000000000002 0000000000000002
Qualidade de Serv.	
- estab_delay	00
- estab_failure	000000
- throughput	001001001001001001001001
- transit_delay	0101010101
- resid_err_rate	111
- Resiliense	000000
- Transf_failure	000000
- Release_delay	00
- Relea_failure	000000
- priority_QOS	0101
- protection	1
Opção D. Expressos	1
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.3 Parâmetros da SP T_Connect_indication

(c') o módulo T_USER[2] envia uma SP **T_Connect_response** (Tabela 6.4) ao módulo TRANSPORT_SERVICE;

- na instância TP2[2], essa SP é recebida pelo módulo ATP que emite, por sua vez, uma TPDU **CC** (Tabela 6.5) ao módulo MAPPING. Os parâmetros da SP T_Connect_response são utilizados para montar essa TPDU CC;

Parâmetro	Conteúdo
Endereço Respond.	00000000000000000000000000000000000002 0000000000000002
Qualidade de Serv.	
- estab_delay	01
- estab_failure	000100
- throughput	001001001001001001001001
- transit_delay	0101010101
- resid_err_rate	111
- Resiliense	000000
- Transf_failure	000000
- Release_delay	01
- Relea_failure	000000
- priority_QOS	0101
- protection	1
Opção D. Expressos	1
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.4 Parâmetros da SP T_Connect_response

Parâmetro	Conteúdo
LI	$(82)_{10}$
CR + CDT	$(11010011)_2$
DST_REF	01
SRC_REF	04
CLASS + OPT	$(00100010)_2$
(parte variável)	
TSAP Chamador	0000000000000001
TSAP Chamado	0000000000000002
TPDU_size	$(00001101)_2$
Version	1
Protection	1
Qualidade de Serv.	
- throughput	001001001001001001001001
- resid_err_rate	111
- priority_QOS	0101
- transit_delay	0101010101
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.5 Parâmetros da TPDU CC

- o módulo MAPPING da instância TP2[2] emite ao módulo NETWORK_SERVICE uma SP **N_Data_request** com uma TPDU CC empacotada no campo dados do usuário;

- o módulo NETWORK_SERVICE envia uma SP **N_DATA_indication** à instância TP2[1];

- na instância TP2[1], o módulo MAPPING decodifica a SP N_Data_indication e emite uma TPDU **CC** ao módulo ATP;

- o módulo ATP da instância TP2[1] emite uma SP T_Connect_confirm (Tabela 6.6) ao módulo T_USER[1]. Os parâmetros dessa SP são obtidos da TPDU CC;

Parâmetro	Conteúdo
Endereço Respond.	00000000000000000000000000000000000000002 0000000000000002
Qualidade de Serv.	
- estab_delay	01
- estab_failure	000100
- throughput	001001001001001001001001
- transit_delay	0101010101
- resid_err_rate	111
- Resiliense	000000
- Transf_failure	000000
- Release_delay	01
- Relea_failure	000000
- priority_QOS	0101
- protection	1
Opção D. Expressos	1
Dados do usuário	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Tabela 6.6 Parâmetros da SP T_Connect_confirm

(d') o módulo T_USER[1] recebe a SP **T_Connect_confirm** do módulo TRANSPORT_SERVICE.

A validação do design do protocolo TP2 consiste na comparação dos traços obtidos na simulação do serviço (Figura 6.15) com os traços obtidos na simulação do protocolo (Figura 6.16). Para um observador externo ao módulo **Transport_Service**, as seqüências de interações, trocadas entre

esse módulo e os módulos **T_User**, devem ser exatamente as mesmas em ambas as simulações. Portanto, nas comparações dos traços, as SPs e TPDUs internas ao módulo Transport_Service da Figura 6.16 devem ser abstraídas.

Em relação ao estabelecimento bem sucedido de uma TC, verifica-se que o traço obtido na simulação do serviço, seqüência (a) (b) (c) (d), é o mesmo obtido na simulação do protocolo, seqüência (a')...(b')...(c')...(d'), desde que as SPs e TPDUs internas ao módulo Transport_Service sejam abstraídas.

Além do estabelecimento bem sucedido de uma TC, vários outros cenários foram simulados:

- estabelecimento de conexão mal sucedido (rejeição pelo usuário par e rejeição pelo provedor);
- transferência de dados normais e transferência de dados expressos;
- desconexão iniciada por um dos usuários e desconexão iniciada pelo provedor.
- etc.

6.4. Geração do Código de Implementação

Nessa fase, procura-se gerar o código de implementação a partir de uma especificação detalhada validada. O objetivo é gerar um código transportável para qualquer ambiente operacional. Uma interface específica simplifica a adaptação do protocolo a cada ambiente operacional.

6.4.1. Adição dos Módulos de Interface

Em relação ao TP2, a arquitetura final, para a geração do código de implementação, é apresentada na Figura 6.17.

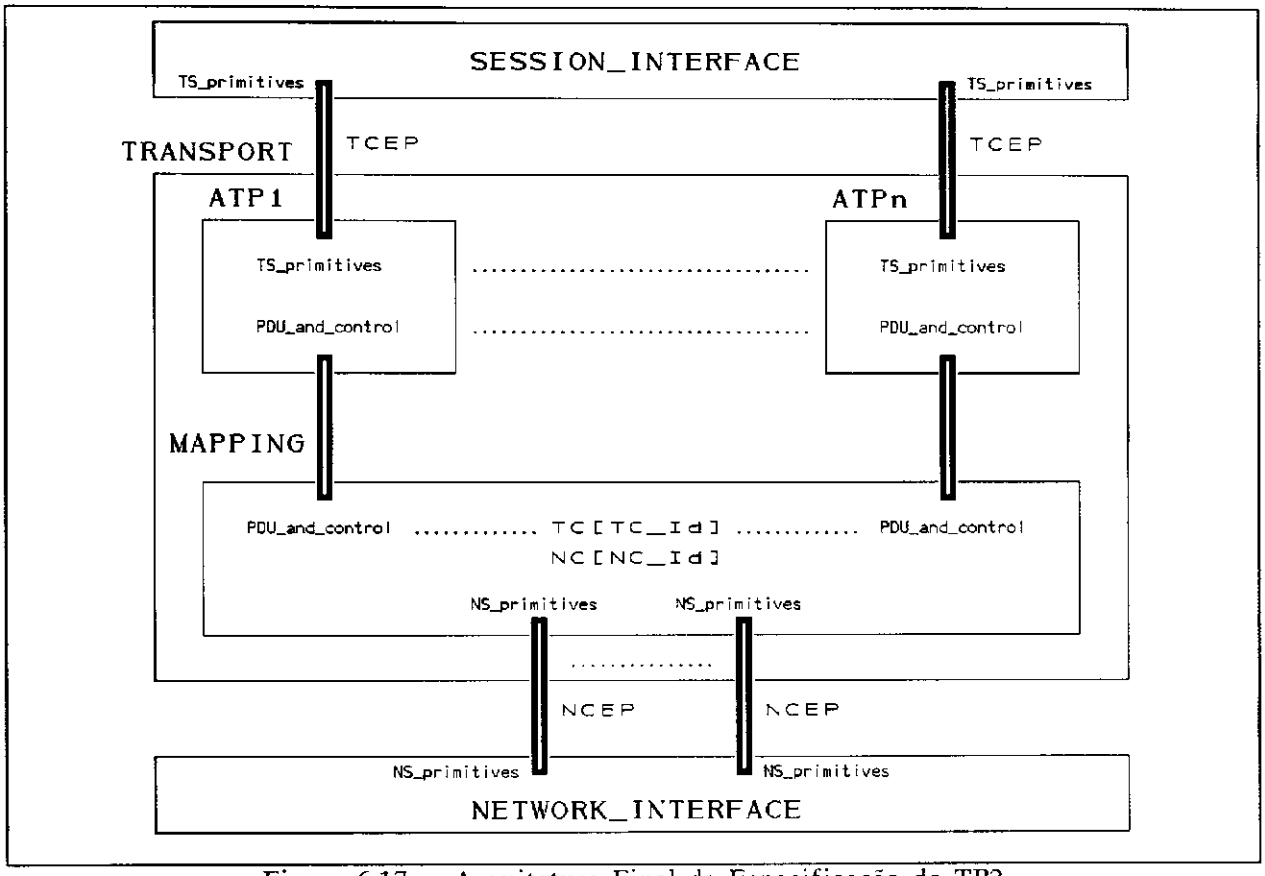

Figura 6.17 Arquitetura Final da Especificação do TP2

A introdução das interfaces entre o TP2 e as camadas adjacentes (Sessão e Rede) permite abstrair os detalhes relativos à implementação dessas camadas. Os módulos **SESSION_INTERFACE** e **NETWORK_ INTERFACE** substituem os módulos **T_USER** e **NETWORK_SERVICE** respectivamente. Na Figura 6.18 é apresentado um fragmento da especificação da interface com a camada de Sessão.

```
01 Any i: TCEP_range do
02 When TSAP[i].T_Connect_Ind(Called_Address, Calling_Address,
03                            Expedited_Data_Option,
04                            Quality_of_Service,
05                            TS_User_data)
06  begin
07   TConInd(i,Called_Address, Calling_Address,
08           Expedited_Data_Option, Quality_of_Service,
09           TS_User_data);
10  end;
11  ***
12  Any i: TCEP_range do
13  provided TConResp_present(i)
14  begin
15    TConResp(Responding_Add, ED_Option, QOS, User_Data);
16    output TSAP[i].T_Connect_Resp(Responding_Add, ED_Option,
17                                  QOS, User_Data);
18  end;
```

Figura 6.18 Fragmento do Módulo SESSION_INTERFACE

Essas interfaces permitem tratar a comunicação com as camadas adjacentes de forma genérica, pois as funções do tipo "primitive" podem ser adaptadas para referenciar essas camadas, independentemente do ambiente operacional e da forma de implementação das mesmas (processo, núcleo do sistema operacional, "firmware", etc). As funções do tipo "primitive" são codificadas na etapa de complementação de código de implementação e ligadas na fase de linkedição.

Essas interfaces podem também transformar a mensagem recebida, que pode estar em qualquer formato, para uma estrutura apropriada definida na especificação.

A especificação final do TP2, contendo também as especificações das interfaces, é submetida ao **EWSTRANS** e o código intermediário é submetido ao **EWSGEN** para a geração do código C orientado para implementação.

6.4.2. Análise do Código Gerado para TP2

O código é gerado à medida que o módulo é declarado na especificação. As estruturas de dados e as funções geradas são codificadas e numeradas de forma crescente, variando essa numeração (###) de 000 a 999. Essa codificação, renomeando os objetos (estruturas de dados e subrotinas) impossibilita o conflito de referências entre os objetos da especificação e os reservados para a implementação.

O código na linguagem C é gerado seguindo a seqüência da própria especificação. Inicialmente, são geradas todas as estruturas de dados correspondentes à parte de declarações da especificação. Em seguida, são geradas as funções e as estruturas de dados que implementam a parte de transições e inicializações dos módulos das especificação. No final do código, são criadas a tabela de funções executáveis, que compõem o corpo de um módulo (_bdesc), e a tabela de descrição dos módulos (_hdesc).

6.4.2.1. Geração das Estruturas de Dados

Os tipos de dados são definidos na linguagem C [KeRi 78] pelo comando "TYPEDEF". As estruturas de dados geradas a partir do fragmento de especificação ilustrado na Figura 6.19 são apresentadas na Figura 6.20.

O valor atribuído às constantes numéricas da especificação (linhas 2 e 14, Figura 6.20) substitui a constante nos locais onde são referenciadas (linhas 6 a 11, Figura 6.19). Um tipo de dado nulo substitui as constantes do tipo caractere (string) nos locais onde são referenciadas.

```
01 Const Max_Data_length        = 16384;
02 ***
03 type
04   Octet = Char;
05   Octet2= array [1 .. 2] of Octet;
06   Octet3= array [1 .. 3] of Octet;
07   ***
08   String_of_octets = record
09      length   : Positive_Integer; (* 0 a 65535 *)
10      contents : array [1 .. Max_Data_length] of Octet;
11   end;
```

Figura 6.19 Trecho de Especificação

```
01 typedef CHAR    Tt000 /* OCTET */ ;
02 typedef CHAR    Tt001 /* OCTET2 */ [2];
03 typedef CHAR    Tt002 /* OCTET3 */ [3];
04 ***
05 typedef CHAR    Tt007 /* CONTENTS */ [16384];
06 /* Type definition : */  /* STRING_OF_OCTETS */
07 typedef struct {
08    INTEGER LENGTH ;
09    Tt007  /* CONTENTS */  /* (Type of) */ CONTENTS ;
10  }Tt008  /* STRING_OF_OCTETS */ ;
```

Figura 6.20 Código da Especificação

As principais estruturas de dados são: Tt### (tipos), Th### (cabeçalho de módulo), Tb### (corpo de módulo) e Tp### (cabeçalho de subrotinas). Elas são analisadas nos itens subseqüentes.

(a) tipo **Tt###**

Os tipos definidos na especificação através do comando **TYPE** são convertidos para estruturas de dados do tipo **Tt###** (onde, ### é um valor entre 000 a 999), conforme as Figuras 6.21 (especificação) e 6.22 (código gerado na linguagem C).

```
01 type
02   Octet = Char;
03   Octet2= array [1 .. 2] of Octet;
04   ***
05   class_type= (undef_class,class_0, class_1, class_2,
06                class_3, class_4);
07   class_set_type = set of class_type;
08   ***
09   Time_type = integer;
10   ***
11   Probability_type = real;
12   ***
13   Option_type = record
14     receipt_confirm, *** : boolean;
15   end;
```

Figura 6.21 Tipos definidos para a especificação

```
01 typedef CHAR    Tt000  /* OCTET */ ;
02 typedef CHAR    Tt001  /* OCTET2 */ [2];
03 ***
04 typedef INTEGER Tt013  /* CLASS_TYPE */ ;
05 ***
06 typedef SET     Tt014  /* CLASS_SET_TYPE */ ;
07 typedef INTEGER Tt018  /* TIME_TYPE */ ;
08 ***
09 typedef REAL    Tt033  /* PROBABILITY_TYPE */ ;
10 ***
11 typedef struct { BOOLEAN RECEIPT_CONFIRM;
12                  *** }Tt045  /* OPTION_TYPE */ ;
13 ***
```

Figura 6.22 Código para os tipos definidos

Os tipos enumerados (linha 5, Figura 6.21) e inteiros (linha 9, Figura 6.21) são convertidos para INTEGER (linhas 4 e 7 da Figura 6.22). Os tipos real (linha 11, Figura 6.21), caractere (linha 2, Figura 6.21), lógico (linha 14, Figura 6.21), conjunto (linha 7, Figura 6.21) e vetor (linha 3, Figura 6.21) são mapeados em tipos correspondentes da linguagem C (linhas 9, 1, 11, 6 e 2 da Figura 6.22, respectivamente).

(b) tipo **Ti###**

Para cada primitiva de interação de um canal (CHANNEL), é gerada uma estrutura de dados do tipo **Ti###** (onde ### é um valor entre 000 a 999). Essa estrutura de dados é composta por:

- uma variável de controle (**_intctrl intctrl**);
- parâmetros (se existirem) trocados na execução de uma interação (**tipovar idvar**, onde **tipovar** é o tipo da variável e **idvar** o identificador da variável).

Por exemplo, para a interação T_Connect_req do canal TS_primitives (Figura 6.23), é gerada a estrutura de dados apresentada na Figura 6.24.

```
channel TS_primitives (T_user, T_provider);
  by T_user :
     T_CONNECT_req    (Called_Address,
                       Calling_Address: TSAP_address_type;
                       Expedited_Data_Option: boolean;
                       Quality_of_Service: QOS_type;
                       TS_User_data: optional_TSDU);
```

Figura 6.23 Especificação da interação T_CONNECT_req

```
01 /* Type definition : */  /* T_CONNECT_REQ */
02 typedef struct {
03   /* Control Data  */
04   _intctrl intctrl ;
05
06   Tt030  /* TSAP_ADDRESS_TYPE */ CALLED_ADDRESS ;
07   Tt030  /* TSAP_ADDRESS_TYPE */ CALLING_ADDRESS ;
08   BOOLEAN EXPEDITED_DATA_OPTION ;
09   Tt043  /* QOS_TYPE */ QUALITY_OF_SERVICE ;
10   Tt008  /* STRING_OF_OCTETS */ TS_USER_DATA ;
11 }Ti000  /* T_CONNECT_REQ */ ;
```

Figura 6.24 Estrutura de dados para interação T_CONNECT_req

(c) tipo **Th###**

Para o cabeçalho de cada módulo (MODULE) é gerada uma estrutura de dados do tipo **Th###** (onde ### é um valor entre 001 a 999). Para a especificação é também gerada uma estrutura de dados **ThSpec** com a mesma estrutura de Th###.

Essa estrutura de dados é composta por (Figura 6.26):

- uma variável de controle (**_ctxctrl ctxctrl**, na linha 4);
- parâmetros passados para inicializar as variáveis do módulo (**tipovar idvar**, após linha 6 se existirem);
- pontos de interação (**_ip idvar**, na linha 9);
- variáveis exportadas (**tipovar idvar**, após a linha 10 se existirem).

Por exemplo, para o cabeçalho do módulo ATP (Figura 6.25), é gerado o código apresentado na Figura 6.26.

```
01 module ATP_ENTITY process;
02  ip TSAP : TS_primitives   (T_Provider) individual queue;
03     MAPX : PDU_and_control (protocol)   individual queue;
04 end (*ATP_ENTITY*);
```

Figura 6.25 Cabeçalho do módulo ATP

```
01 /* Type definition : */  /* ATP_ENTITY */
02 typedef struct {
03   /* Control Data */
04   _ctxctrl ctxctrl ;
05
06   /* Parameters */
07
08   /* External IP and Exported Data */
09   _ip TSAP ;
10   _ip MAPX ;
11 }Th004  /* ATP_ENTITY */ ;
```

Figura 6.26 Código para cabeçalho do módulo ATP

(d) tipo **Tb###**

Para o corpo de cada módulo (BODY) (Figura 6.27) é gerada uma estrutura de dados do tipo **Tb###** (onde ### é um valor entre 000 a 999). Essa estrutura de dados (Figura 6.28) é composta por:

- cabeçalho do módulo (linha 3), contendo as variáveis de controle e as variáveis para comunicação com o exterior (parâmetros, pontos de interação e variáveis exportadas);

- variáveis internas, isto é, canais internos (linhas 6 e 7), variáveis internas ao módulo (linhas 9 a 26), variáveis de estado (linha 8), variáveis para criação de instâncias de um de módulo (linhas 27 e 28);

- tabelas de transições (linhas 30 a 32), contendo as variáveis indexadas **TabTrans** (tabela de transições disparáveis) e **TchTrans** (tabela de transições nunca disparadas);

```
01 body ATP_ENTITY_BODY for ATP_ENTITY;
02   channel T_Clock_Access (User,Provider);
03   ***
04   State CLOSED, NET_OPEN_IN_PROGRESS, ***
05   ***
06   var
07    Class    : class_type;
08    TR,
09    TS   : seq_number_type;  (* num. de sequencia *)
10    R_Credit   : credit_type;
11    S_Credit   : credit_type;
12    Receive_buffer: data_buffer;
13    Send_buffer    : data_buffer;
14    ***
15    Modvar
16      CLOCK1, CLOCK2 : T_CLOCK;
17    ***
18    initialize to CLOSED
19      begin ***
20    ***
```

Figura 6.27 Trecho da especificação do corpo do módulo ATP

- tabela de inicializações (linha 35), contendo a variável indexada **TabInit;**

- tabela de transições espontâneas (se existirem), contendo as variáveis indexadas (**AnyRes** e **AnyPt**) com 1000 elementos alocados previamente pelo gerador de códigos.

```
01   /* ATP_ENTITY_BODY */ typedef struct {
02   /* External and Control Data */
03   Th004  /* ATP_ENTITY */ Ext ;
04
05   /* Internal Data */
06   _ip TIMER1 ;
07   _ip TIMER2 ;
08   ENTRYINDEX MajorState ;
09   INTEGER CLASS ;
10   INTEGER TR ;
11   INTEGER TS ;
12   INTEGER R_CREDIT ;
13   INTEGER S_CREDIT ;
14   Tt061  /* DATA_BUFFER */ RECEIVE_BUFFER ;
15   Tt061  /* DATA_BUFFER */ SEND_BUFFER ;
16   BOOLEAN ED_SEND ;
17   BOOLEAN ED_RECEIVED ;
18   BOOLEAN PROTOCOL_ERROR_STATE ;
19   Tt051  /* TPDU_AND_CONTROL_INFORMATION */ LOCAL_TPDU ;
20   Tt043  /* QOS_TYPE */ QTS_IND_S ;
21   BOOLEAN ED_OPTION_S ;
22   INTEGER TPDU_SIZE_S ;
23   INTEGER DT_UD_LENGTH ;
24   BOOLEAN EXT_FORMAT ;
25   BOOLEAN NO_FLOW_CONT ;
26   Tt008  /* STRING_OF_OCTETS */ NULL_DATA ;
27   _modvar CLOCK1 ;
28   _modvar CLOCK2 ;
29
30    /* Transitions Table */
31   _tt TabTrans [58];
32   INTEGER TchTrans [58];
33
34    /* Initialize Table */
35   _tt TabInit [2];
36
37 }Tb004  /* ATP_ENTITY_BODY */;
```

Figura 6.28 Estrutura de dados para o Corpo de módulo

(e) tipo **Tp###**

Para o cabeçalho de cada "FUNCTION" ou "PROCEDURE" (Figura 6.29) é gerada uma estrutura de dados do tipo **Tp###** (onde ### é um valor entre 000 a 999). Essa estrutura de dados é composta por (Figura 6.30):

- variáveis do módulo (linha 4);

- parâmetros da subrotina original (linhas 7 a 9);

- variáveis locais da subrotina (linhas 12 e 13).

```
01 (* testa se pode incrementar creditos *)
02 function Credit_Increm (Buffer: data_buffer;
03                         Credit:credit_type;
04                         Max_length:integer): boolean;
05  var I, J:Integer;
06  begin
07    J:=0; (* block counter *)
08    for I:=1 to Nr_Buffer_Block do
09      if  Buffer.space[I]=0 then J:=J+1;
10    if (J > Credit) then Credit_Increm:=true;
11  end;
```

Figura 6.29 Subrotina Credit_Increm

```
01 /* Type definition : */  /* CREDIT_INCREM */
02 typedef struct {
03   /* Upper Data Link */
04   Tb004  /* ATP_ENTITY_BODY */ * Loc ;
05
06   /* Parameters */
07   Tt061  /* DATA_BUFFER */ BUFFER ;
08   INTEGER CREDIT ;
09   INTEGER MAX_LENGTH ;
10
11   /* Local Variables */
12   INTEGER I ;
13   INTEGER J ;
14 }Tp031  /* CREDIT_INCREM */ ;
```

Figura 6.30 Estrutura de dados para as subrotinas

6.4.2.2. Geração das funções

Esse segundo bloco contém as funções e estruturas de dados internas, que implementam as subrotinas, as transições e a parte de inicializações da especificação.

Uma numeração crescente (variando de 000 a 999) é associada a cada módulo para compor o nome das funções especiais (TrEvb###, Trb###, InEvb###, InExb###, Gub###, Inb### e Ihbb###) e estruturas de dados (Anyb###, TrTab###, Whenb### e Fromb###). Uma outra numeração crescente (variando de 000 a 999) também é associada a cada subrotina (p###).

As funções geradas por EWSGEN são criadas de acordo com a ordem em que os módulos são definidos.

Os comandos em Pascal são convertidos para os correspondentes em C. Algumas funções especiais são utilizadas para atribuição e comparação, tais como: afinterv (atribui valores às variáveis do tipo enumerado), set_vaffect (realiza a união de variáveis tipo "SET"), set_init (inicializa as variáveis do tipo "SET"), set_in (verifica se um elemento faz parte do conjunto), set_add(adiciona um elemento ao conjunto), cpzone (compara variáveis), cpbool (compara variáveis lógicas), TestIndex (testa a validade do índice de variáveis indexadas), TestZero (verifica se o divisor é zero), etc.

Em resumo, os componentes executáveis de uma especificação são traduzidos da seguinte forma:

- as subrotinas são traduzidas para funções p###;
- a parte responsável pela inicializações do corpo do módulo (INITIALIZE) é a função Inb### (que ativa as funções InEvb### e InExb###);

- a parte responsável pela habilitação de uma transição (WHEN/PROVIDED) é traduzido para a função Gub### (que chama a função TrEvb###);

- a parte responsável pela execução da transição é implementado pela função Trb###.

As funções geradas por EWS para implementar as subrotinas, a parte de inicializações e as transições são apresentadas nos itens subseqüentes.

(a) funções **p###**

As funções **p###**, que implementam as subrotinas originais da especificação (PROCEDURE/FUNCTION), são numeradas de forma crescente e global, variando ### de 000 a 999. As subrotinas vão sendo convertidas à medida que aparecem na especificação.

As primeiras subrotinas a serem convertidas são as do tipo **primitive**, pois elas são declaradas a nível global. Somente o cabeçalho da função **p###** é gerado (declaração da função e parâmetros). O código dessas subrotinas deve ser codificado à parte e ligado posteriormente ao código da especificação.

As subrotinas oriundas de "procedure" (Figura 6.31) são convertidas em funções sem tipo. Temos os seguintes parâmetros para essas funções (Figura 6.32):

- parâmetros da subrotina original (linhas 3 e 4);

- a variável **locup** do tipo "Tb###", contendo todas as estruturas de dados do módulo a serem utilizadas na subrotina (linha 5).

```
01 procedure protocol_error
02   (var TPDU:TPDU_and_control_information;
03        cause : reject_cause_type);
04   begin
05     with TPDU do
06       begin
07         kind := ER;
08         reject_cause := cause;
09       end;
10     Protocol_Error_State:=True;
11   end;
```

Figura 6.31 Exemplo de Procedimento

```
01 p017  /* PROTOCOL_ERROR */ (TPDU ,CAUSE ,locup)
02
03   Tt051  /* TPDU_AND_CONTROL_INFORMATION */ *TPDU ;
04   INTEGER CAUSE ;
05   Tb004  /* ATP_ENTITY_BODY */  * locup;
06
07   { mp17 /* Variables intermediaires */
08     Tp017  /* PROTOCOL_ERROR */ Loc;
09
10     Loc.TPDU  = TPDU ;
11     Loc.CAUSE  = CAUSE ;
12     Loc.Loc = locup;
13     {
14      {Tt051 *  withvar0;
15        withvar0 = & ((*(Loc.TPDU )) ) ;
16        {withvar0->KIND  = ( 9 );
17         withvar0->REJECT_CAUSE  = Loc.CAUSE ;
18        }
19      }
20      Loc.Loc->PROTOCOL_ERROR_STATE  = ( 1 );
21     }
22   }
```

Figura 6.32 Código gerado para procedimento PROTOCOL_ERROR

As subrotinas oriundas de "function" (Figura 6.33) são convertidos em funções com o tipo correspondente ao original. Temos os seguintes parâmetros para essas funções (Figura 6.34):

- parâmetros da subrotina original (linhas 3 a 5);
- a variável **locup** do tipo "Tb###", contendo todas as estruturas de dados do módulo a serem utilizadas na subrotina (linha 6);

- a variável **retour** (linha 10) é adicionada às subrotinas do tipo "function", contendo o valor a ser retornado pela função original.

```
01 (* testa se pode incrementar creditos *)
02 function Credit_Increm(Buffer: data_buffer;
03                        Credit:credit_type;
04                        Max_length:integer): boolean;
05  var I, J:Integer;
06  begin
07    J:=0; (* block counter *)
08    for I:=1 to Nr_Buffer_Block do
09      if  Buffer.space[I]=0 then J:=J+1;
10      if (J > Credit)
11        then
12          Credit_Increm:=true;
13  end;
```

Figura 6.33 Exemplo de função

```
01 BOOLEAN p031  /* CREDIT_INCREM */ (BUFFER ,CREDIT,
02                                    MAX_LENGTH ,locup)
03   Tt061  /* DATA_BUFFER */ *BUFFER ;
04   INTEGER CREDIT ;
05   INTEGER MAX_LENGTH ;
06   Tb004  /* ATP_ENTITY_BODY */  * locup;
07   {
08     mp31 /* Variables intermediaires */
09     Tp031  /* CREDIT_INCREM */ Loc;
10     BOOLEAN retour;
11
12     cpzone(&(Loc.BUFFER ), BUFFER , sizeof(Loc.BUFFER ) );
13     Loc.CREDIT  = CREDIT ;
14     Loc.MAX_LENGTH  = MAX_LENGTH ;
15     Loc.Loc = locup;
16     {
17       Loc.J  = ( 0 );
18       ForUp( Loc.I , ( 1 ), ( 3 ) )
19        {
20        if (( Loc.BUFFER .SPACE [TestIndex(Loc.I ,1,3)] ==
21           ( 0 ) ))
22             {Loc.J  = ( Loc.J  + ( 1 ) );}
23        }
24       if (( Loc.J  > Loc.CREDIT  ))
25          {retour = ( 1 );}
26     }
27     return(retour);
28   }
```

Figura 6.34 Código gerado para uma função

A variável **Loc** do tipo "Tp###" (linha 9, Figura 6.34) contém todas as estruturas de dados locais ao módulo e à subrotina e contém parâmetros da subrotina original.

(b) tabela ANY (**Anyb###**)

A tabela ANY (variável **Anyb###**, onde ### é a numeração associada ao módulo) representa todas as transições espontâneas do módulo (linhas 6 a 9, Figura 6.35). A quantidade de elementos na tabela ANY é definida pela quantidade de transições espontâneas.

```
01 static INTEGER A1029t1b6 [] = {1029, /* Interv */ 2,1,4,
02        NULL };
03 static INTEGER *  Any1b6 [] = {A1029t1b6 ,  NULL };
04 ***
05 /* ANY TABLE  */
06 static INTEGER **  Anyb006  /* MAPPING_BODY */  [] =
07 { NULL ,Any1b6, Any2b6, Any3b6, Any4b6, Any5b6, Any6b6,
08   Any7b6, Any8b6, Any9b6, Any10b6, Any11b6, Any12b6,
09   Any13b6, Any14b6, Any15b6,};
```

Figura 6.35 Tabela das transições espontâneas

A variável **Anyb###** é uma variável indexada que contém as variáveis **Any?b$** (e.g., Any1b6, ..., Any15b6). A variável **Any?b$** (linha 3, Figura 6.35) representa cada transição espontânea. O nome da variável **Any?b$** é obtido da seguinte forma: **Any?** representa a numeração associada à transição (n-ésima transição espontânea do módulo) e **b$** representa a numeração associada ao módulo.

A variável **Any?b$** contém as variáveis **A&t?b$** (e.g., A1029t1b6). A variável **A&t?b$** (linhas 1 e 2 da Figura 6.35) representa cada variável da cláusula **ANY**. O nome da variável **A&t?b$** é obtido da seguinte forma: **A&** representa a numeração global dada a variável da cláusula ANY, **t?** representa a numeração associada à transição (n-ésima transição espontânea do módulo) e

b$ é a numeração associada ao módulo. Essa variável contém a numeração dada à variável (por exemplo, 1029) e o intervalo de variação da variável ANY (por exemplo, variando de 1 a 4).

(c) função **TrEvb###**

Ativada pela função Gub###, a função **TrEvb###**, onde ### é a numeração dada ao módulo, contém a parte responsável pela habilitação das transições de um módulo, correspondente às cláusulas "PROVIDED" e "WHEN".

Os parâmetros dessa função são (linhas 4 a 13, Figura 6.37): a variável de controle (**CTX_P**), o número da transição (**NoTrans**) e o índice para a cláusula ANY (**IndexAny_p**).

A variável **_eval** é inicializada com 1 (ou seja, "true"). As variáveis **Int** e **Ext** contém as variáveis do módulo e as variáveis das interfaces do módulo respectivamente.

Para a cláusula **WHEN** (linha 2, Figura 6.36) é gerada uma chamada à função de suporte **WHENINTER** (linhas 21 a 23, Figura 6.37) para avaliar a existência de uma interação.

Os parâmetros de uma interação (linha 2, Figura 6.36) são obtidos com a atribuição da variável de interface "EXT" à variável **ip_p** (linha 24, Figura 6.37).

Para a cláusula **PROVIDED** (linha 3, Figura 6.36) é gerada uma atribuição da expressão booleana à variável **_eval** (linha 26, Figura 6.37). Nesse caso, a expressão booleana é p016, que corresponde à função booleana "IN_USE" da especificação.

A variável **_eval** é retornada, habilitando (ou não) a transição (linha 28, Figura 6.37).

```
01 trans
02  when TSAP.T_CONNECT_req (Called_Address, ***
03  provided not In_Use
04  from CLOSED to OPEN_IN_PROGRESS_CALLING
05  begin
06    TR := 1;
07    TS := 0;
08    ED_send := false;
09    ED_received := false;
10    S_Credit:=0;
11    R_Credit:=Nr_Buffer_block;
12    Class:=Implemented_Class(Quality_of_Service);
13    QTS_ind_s:=Quality_of_Service;
14    ED_option_s:=Expedited_Data_Option;
15    local_TPDU.kind:=CR;
16    with local_TPDU do
17     begin
18       Kind := CR;
19       credit_value:=R_Credit;
20       Called_Addr:= Called_address;
21       Calling_Addr:=Calling_Address;
22       Options.Non_use_of_explicit_flow_control:=
23                                    No_flow_control1;
24        Options.extended_format:=extended_format1;
25        QTS_ind := Quality_of_Service;
26        Class_Ind:=Class;
27        Alternate_Class:=Alt_Class_Set;
28        TPDU_Size_ind:=TPDU_size_proposed;
29        ED_option:=Expedited_Data_Option;
30        if (TS_User_data.length > T_CONN_UD_length)
31          then
32            begin
33              (* data length greater than *)
34              User_Data:=NULL_DATA;
35           end
36          else
37            User_Data:=TS_User_Data;
38     end;
39    output Timer1.T_Set_Clock(
40           Quality_of_Service.Establishment_delay);
41    output MAPX.dados (local_TPDU);
42  end;
```

Figura 6.36 Fragmento de Transição

```
01 /* Transition Evaluation */
02 static TrEvb004  /* ATP_ENTITY_BODY */ (ctx_p, NoTrans,
03                                          IndexAny_p)
04  _ctxctrl * ctx_p      ;
05  INTEGER    NoTrans    ;
06  INTEGER  * IndexAny_p ;
07  {Tb004  /* ATP_ENTITY_BODY */ * Int ;
08   Th004  /* ATP_ENTITY */ * Ext ;
09   INTEGER _eval;
10   _ip   * ip_p ;
11   _eval = 1 ;
12   Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
13   Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
14   SETJMP ;
15
16   switch(NoTrans)
17   {
18    ***
19    case 2 :
20     ip_p = &(Ext->TSAP );
21     if (_eval) _eval = _wheninter(ctx_p,
22                                   &(Ext->TSAP ),
23                                   0  /* T_CONNECT_REQ */ );
24     ip_p = &(Ext->TSAP);
25     {mc13 /* Variables intermediaires */
26         if (_eval) _eval = !(p016  /* IN_USE */ (ctx_p));
27     };
28     return(_eval);
29 ***
```

Figura 6.37 Código da função TrEvb###

(d) função **Trb###**

A função **Trb###** (onde ### é a numeração dada ao módulo) contém a parte de "ação" de uma transição, implementando a transição de estados (cláusula TO) e o bloco de comandos, que contém a atualização das variáveis e a geração de interações de saída.

O parâmetro da função é a variável de controle **CTX_P**. Inicialmente é feita uma avaliação para obter uma transição disparável (linhas 14 a 25, Figura 6.38). A variável **NoTrans** indica a transição a ser disparada. As variáveis **Int** e **Ext**

(linhas 11 e 12, Figura 6.38) contém as variáveis do módulo e de interface respectivamente. A obtenção dos parâmetros da interação de entrada é feita pela função **_whenget** (linha 31, Figura 6.38).

A mudança de estado (linha 4, Figura 6.36) é feito através da atribuição de um valor à variável de estado **MajorState** (linha 30, figura 6.38).

Em relação ao bloco de comandos, ele pode conter comandos Pascal (atribuição de variáveis, comando WITH, chamada a subrotinas, etc) e comandos Estelle (comando OUTPUT, INIT, RELEASE, CONNECT, DISCONNECT, ATTACH, DETACH, ALL/FORONE, ANY, EXPORT, etc).

A atribuição de variáveis é feita com o auxílio de funções que testam a validade dessa atribuição, tais como, cpzone (linha 53, Figura 6.38). As chamadas às subrotinas são traduzidas pela ativação das funções p### (linha 40 a 45, Figura 6.38).

O comando **WITH** (linha 16, Figura 6.36) é traduzido na declaração e na atribuição de valor à variável **WITHVAR#** (linhas 49 e 50, Figura 6.38). Uma numeração é atribuída à variável WITHVAR# (withvar0, withvar1, ...), permitindo vários níveis para o comando WITH.

O comando **OUTPUT**, que gera as interações de saída (linha 39 a 41, Figura 6.36), é convertido para as funções **_mkinter, _outargv** e **_outargv** (linhas 60 a 65 e linhas 67 a 69, Figura 6.38). A função _mkinter indica o tipo de interação no canal. A função _outargv permite associar o parâmetro da interação de saída ao parâmetro informado no comando OUTPUT . A função _output gera a interação de saída.

```
01 /* Transition Execution */ /* +FC */
02 Trb004  /* ATP_ENTITY_BODY */ (ctx_p)
03  _ctxctrl * ctx_p      ; {
04  {INTEGER    NoTrans    ;
05   INTEGER  * IndexAny_p ;
06   Tb004  /* ATP_ENTITY_BODY */ * Int ;
07   Th004  /* ATP_ENTITY */ * Ext ;
08   INTEGER _eval;
09   _ip   * ip_p ;
10   _eval = 1 ;
11   Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
12   Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
13
14   if ((ctx_p->_SimCtx.NoFireTrans)<(Int->TabTrans[0].num))
15       (ctx_p->_SimCtx.NoFireTrans)  ++  ;
16   else (ctx_p->_SimCtx.NoFireTrans)  = 1 ;
17   NoTrans = (ctx_p->_SimCtx.NoFireTrans);
18   if (Int->TabTrans[NoTrans].AnyVal)
19   { int i;
20     for(i=0;Int->TabTrans[NoTrans].AnyVal[i];i++)
21      if (i > (ctx_p->_SimCtx.NoFireAny))  break ;
22     if (!(Int->TabTrans[NoTrans].AnyVal[i]))  i = 0 ;
23     (ctx_p->_SimCtx.NoFireAny) = i ;
24     IndexAny_p = Int -> TabTrans[NoTrans].AnyVal[i]; }
25   NoTrans = (Int->TabTrans[NoTrans].num);
26   SETJMP ;
27   switch(NoTrans) {
28   ****
29   case 2 :
30    Int->MajorState= 487  /* OPEN_IN_PROGRESS_CALLING */ ;
31    _whenget(ctx_p, &(Ext->TSAP ));
32    {mt22 /* Variables intermediaires */
34      {Int->TR  = ( 1 );
35       Int->TS  = ( 0 );
36       Int->ED_SEND  = ( 0 );
37       Int->ED_RECEIVED  = ( 0 );
38       Int->S_CREDIT  = ( 0 );
39       Int->R_CREDIT  = ( 3 );
40       Int->CLASS = p020  /* IMPLEMENTED_CLASS */
41            (&(_inarg(ctx_p,Ti000  /* T_CONNECT_REQ */,
42               QUALITY_OF_SERVICE)), ctx_p);
43       cpzone(&(Int->QTS_IND_S ), &( _inarg(ctx_p,Ti000
44              /* T_CONNECT_REQ */ ,QUALITY_OF_SERVICE )),
45               sizeof(Int->QTS_IND_S ));
46       Int->ED_OPTION_S  = _inarg(ctx_p,Ti000
47              /* T_CONNECT_REQ */ ,EXPEDITED_DATA_OPTION );
48       Int->LOCAL_TPDU .KIND  = ( 1 );
```

(continua na próxima página)

```
49        {Tt051 *  withvar0;
50             withvar0 = & (Int->LOCAL_TPDU ) ;
51             { withvar0->KIND  = ( 1 );
52                withvar0->CREDIT_VALUE  = Int->R_CREDIT ;
53                cpzone(&(withvar0->CALLED_ADDR ),
54                  &(_inarg(ctx_p,Ti000 /* T_CONNECT_REQ */,
55                  CALLED_ADDRESS)),
56                  sizeof(withvar0->CALLED_ADDR ));
57             ***
58
59        }
60        _mkinter(ctx_p, Ti032  /* T_SET_CLOCK */ ,
61             32  /* T_SET_CLOCK */ );
62        _outargv(ctx_p, Ti032  /* T_SET_CLOCK */ ,
63             DELAY_TIME,_inarg(ctx_p,Ti000 /*T_CONNECT_REQ*/,
64             QUALITY_OF_SERVICE ).ESTABLISHMENT_DELAY);
65        _output(ctx_p, &(Int->TIMER1 ));
66
67        _mkinter(ctx_p, Ti029 /*DADOS*/, 29 /* DADOS */ );
68        _outargr(ctx_p,Ti029/*DADOS*/,TPDU,Int->LOCAL_TPDU);
69        _output(ctx_p, &(Ext->MAPX ));
70      }
71    }
72   _whenforget(ctx_p);
73   return(_eval);
74 ***
```

Figura 6.38 Código da função Trb###

A criação de instâncias de um módulo pode ser estática (dentro de uma cláusula INITIALIZE) ou dinâmica (dentro de uma transição). O comando **INIT** é traduzido para as funções **mkctx**, **Ihbb###** e **_runinit**.

A cláusula ANY é implementada da seguinte forma:

- inicialmente é avaliada a tabela de transições espontâneas (linhas 18 a 24, Figura 6.39);

- a instância do módulo a ser utilizada é indicada pelo valor da variável **IndexAny_p[1]** (linha 13, Figura 6.39).

```
01  (* especificação para a cláusula ANY *)
02  ***
03  any i: NCEP_range do
04  when NSAP[i].N_CONNECT_req ( ***
05  provided not (NET_error(i))
06  begin
07  output NSAP[destination(i)].N_CONNECT_Ind (***
08
09  /* código para a cláusula ANY */
10  ***
11  case 1 :
12  _whenget(ctx_p,
13    &(Ext->NSAP[TestIndex((IndexAny_p[1]) /* I */,1,2)]));
14    {mt12 /* Variables intermediaires */
15      {_mkinter(ctx_p,Ti018 /* N_CONNECT_IND */,
16              18 /* N_CONNECT_IND */ );
17              _outarga(ctx_p, Ti018  /* N_CONNECT_IND */ ,
18                      CALLED_ADDRESS , _inarg(ctx_p,Ti010
19                      /* N_CONNECT_REQ */, CALLED_ADDRESS));
20      ***}
21      _whenforget(ctx_p);}
22      return(_eval);}
```

Figura 6.39 Especificação e código para a cláusula ANY

A exportação de variáveis do módulo (EXPORT) é feito atribuindo-se um valor à variável exportada e ativando-se a função _ctxpar, que avisa se o parâmetro foi alterado (linhas 1 a 3, Figura 6.40). A obtenção do valor da variável exportada é feito pela utilização das funções **_expvarv** ou **_exparva** (linha 1, Figura 6.40).

```
01  (* atribuicao da variavel exportada na especificacao *)
02  NUMERO_CHAMADO=CONTROLADOR.NRO_CHAMADO
03  ***
04
05  /* codigo gerado */
06  Ext->NUMERO_CHAMADO = _expvara(Int->CONTROLADOR, Th006
07                        /* TIPO_CONTROLADOR */, NRO_CHAMADO)
08  _ctxpar(ctx_p);
```

Figura 6.40 Especificação e Código para a Exportação de Variáveis

(e) função **InEvb###**

Ativada pela função Inb###, a função **InEvb###**, onde ### é a numeração dada ao módulo, é responsável pela avaliação da cláusula **PROVIDED** da parte de inicializações de um módulo (comando INITIALIZE).

Os parâmetros desta função (linhas 4 a 6, Figura 6.41) são a variável de controle (**CTX_P**), o número da transição a ser avaliada (**NoTrans**) e o índice para a cláusula ANY (**IndexAny_p**).

O resultado da avaliação da expressão booleana da cláusula **PROVIDED** é atribuído à variável **_eval** (linha 16, Figura 6.41) e retornado pela função InEvb###.

```
01 /* Initialization Evaluation */
02 static InEvb004 /*ATP_ENTITY_BODY*/(ctx_p, NoTrans,
03                                      IndexAny_p)
04 _ctxctrl * ctx_p      ;
05 INTEGER    NoTrans    ;
06 INTEGER  * IndexAny_p ;
07 {Tb004  /* ATP_ENTITY_BODY */ * Int ;
08  Th004  /* ATP_ENTITY */ * Ext ;
09  INTEGER _eval;
10  _ip   * ip_p ;
11  _eval = 1 ;
12  Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
13  Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
14  SETJMP ;
15  switch(NoTrans)
16   {case 1 :  return(_eval);} /* switch */
17  return(_eval);
18 } /*End of procedure definition*/  /* ATP_ENTITY_BODY */
```

Figura 6.41 Código da função InEvb004

(f) função **InExb###**

A função **InExb###**, onde ### é a numeração dada ao módulo, implementa a parte de inicializações do corpo de um módulo, que contém o comando INITIALIZE e o bloco de comandos associado (linhas 4 a 16, Figura 6.42).

```
01  Modvar
02    CLOCK1, CLOCK2 : T_CLOCK;
03
04  initialize to CLOSED
05    begin
06      (* instancia o temporarizador *)
07      init CLOCK1 with T_CLOCK_BODY;
08      init CLOCK2 with T_CLOCK_BODY;
09      connect TIMER1 to CLOCK1.C_PORT;
10      connect TIMER2 to CLOCK2.C_PORT;
11
12      Protocol_Error_State:=False;
13      NULL_DATA.length:=0;
14      Init_buffer(Receive_buffer);
15      ***
16    end;
17  ***
```

Figura 6.42 fragmento para inicialização de módulo

Os parâmetros da função (linhas 4 a 6, Figura 6.43) são: a variável de controle (**CTX_P**), o número da transição a ser avaliada (**NoTrans**) e o índice para a cláusula ANY (**IndexAny_p**). As variáveis **Int** e **Ext** (linhas 12 e 13, Figura 6.43) contém as variáveis do módulo e as variáveis de interface respectivamente.

O estado inicial da MEF (linha 4, Figura 6.40) é estabelecido através da atribuição de um valor à variável de estado **MajorState** (linha 17, Figura 6.43).

A criação das instâncias de um módulo (linhas 7 e 8, Figura 6.42) é realizada da seguinte forma:

```
01 /* Initialisation Execution */
02 static InExb004 /*ATP_ENTITY_BODY*/ (ctx_p, NoTrans,
03                                       IndexAny_p)
04 _ctxctrl * ctx_p      ;
05 INTEGER    NoTrans    ;
06 INTEGER  * IndexAny_p ;
07 {Tb004  /* ATP_ENTITY_BODY */ * Int ;
08  Th004  /* ATP_ENTITY */ * Ext ;
09  INTEGER _eval;
10  _ip   * ip_p ;
11  _eval = 1 ;
12  Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
13  Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
14  SETJMP ;
15  switch(NoTrans)
16  {case 1 :
17   Int->MajorState  = 485  /* CLOSED */ ;
18   {mt78 /* Variables intermediaires */
19    {_mkctx(ctx_p, &(Int->CLOCK1),Tb005  /*T_CLOCK_BODY*/ ,
20       5 /* T_CLOCK */, 5  /* T_CLOCK_BODY */ );
21     Ihbb005 /*T_CLOCK_BODY*/ (_modctx(&(Int->CLOCK1 )),0);
22     _runinit(Int->CLOCK1 ,5  /* T_CLOCK_BODY */ );
23     SETJMP ;
24
25     _mkctx(ctx_p,&(Int->CLOCK2),Tb005 /*T_CLOCK_BODY*/,
26        5  /* T_CLOCK */, 5  /* T_CLOCK_BODY */ );
27     Ihbb005 /*T_CLOCK_BODY*/ (_modctx(&(Int->CLOCK2 )),0);
28     _runinit(Int->CLOCK2 ,5  /* T_CLOCK_BODY */ );
29     SETJMP ;
30
31     _connect(ctx_p,&(Int->TIMER1),_modctx(&(Int->CLOCK1 )),
32        &(_eipvar(_modctx(&(Int->CLOCK1)),Th005 /*T_CLOCK*/,
33        C_PORT )));
34     _connect(ctx_p,&(Int->TIMER2),_modctx(&(Int->CLOCK2 )),
35        &(_eipvar(_modctx(&(Int->CLOCK2)),Th005,C_PORT)));
36     Int->PROTOCOL_ERROR_STATE  = ( 0 );
37     afinterv(Int->NULL_DATA.LENGTH, (0), 0, 32767, 1242 );
38     p023  /*INIT_BUFFER*/ (&(Int->RECEIVE_BUFFER ), ctx_p);
39    ***
40    }
41    return(_eval);
42   } /* switch */ ;
43   return(_eval);
44  } /*End of procedure definition*/  /*ATP_ENTITY_BODY*/
```

Figura 6.43 Código da função InExb004(inicialização da execução)

- a função **_mkctx** é ativada (linha 19, Figura 6.43), criando-se a instância do módulo;

- a função **Ihbb###** é ativada (linha 21, Figura 6.43), inicializando-se o corpo do módulo;

- a função **_runinit** é ativada (linha 22, Figura 6.43), inicializando-se o módulo.

A conexão entre pontos de interação de módulos é realizada através do comando **CONNECT** (linha 9, Figura 6.42). Esse comando é implementado ativando-se a função **connect** (linhas 31 a 33, Figura 6.43).

A vinculação entre pontos de interação de um módulo pai e pontos de interação de seus filhos é realizada através do comando ATTACH (linha 5, Figura 6.44). A função **attach** (linhas 13 a 15, Figura 6.44) é utilizada para implementar esse comando.

O comando **ALL** (linha 1, Figura 6.44) é traduzido para um laço de iteração (linhas 9 a 17, Figura 6.44) associado às funções InitEnum e GetEnum (linhas 10 e 11, Figura 6.44). Após a declaração da variável de iteração (linha 9, Figura 6.44), **InitEnum** inicializa essa variável e **GetEnum** verifica se o limite superior da iteração foi atingido.

```
01 all i:TCEP_range do
02  begin
03    init ATP[i] with ATP_ENTITY_BODY;
04    connect ATP[i].MAPX to MAP.ATPX[i];
05    attach  TSAP[i] to ATP[i].TSAP;
06  end;
07
08
09  {INTEGER I ;
10   InitEnum(1078, /* Interv */ 2,1,4);
11   while (GetEnum(1078 , & I ))
12    {***
13     attach(ctx_p, &(Ext->TSAP [TestIndex(I ,1,4)]),
14          modctx(&(Int->ATP [TestIndex(I ,1,4)])),
15          &(_eipvar(_modctx(&(Int->ATP[TestIndex(I ,1,4)])),
16          Th004  /* ATP_ENTITY */ ,TSAP )));
17     ***}}
```

Figura 6.44 Fragmento da especificação e do código gerado para o comando ATTACH

(g) Tabela de Transição (**TrTab###**), Tabela WHEN (**Whenb###**) e Tabela FROM (**Fromb###**)

As tabelas são declaradas e inicializadas nessa ordem:

- é criada uma instância da constante **NuTrb###**, que indica a quantidade de transições do módulo (linha 3, Figura 6.45);

- um valor é atribuído à variável **NuInb###**, indicando a existência da parte de inicializações do corpo de um módulo (linha 13, Figura 6.42);

```
01  /* Transition Table */
02  /* Nb of Transition */
03  static INTEGER NuTrb004  /*  ATP_ENTITY_BODY */  = 57;
04  /* Transition Part Table  */
05  static _tTT TrTab004  /* ATP_ENTITY_BODY */ [58] =
06    { /* No Type Ident To   Prov  Any  Prio  Delay */
07       { 0 ,  0,   0 ,  0 ,  0  , '0' ,  0  , NULL },
08       { 2 ,  9 ,  0 , 487, 'p' , '0' , -1 ,  NULL }
09       ***}
10
11  /* Initialize Transition Table */
12  /* Nb of Initialize Transition  */
13  static INTEGER NuInb004  /* ATP_ENTITY_BODY */  = 1;
14
15  /* Initialize Part Table  */
16  static _iTT InTab004  /* ATP_ENTITY_BODY */ [2] = {
17
18  /*  No  Type   Ident  To  Prov   Any  Prio  Delay */
19    { 0 ,  0  ,   0  ,  0  ,  0  ,  '0' ,  0  , NULL },
20    { 1 , 20 ,    0  , 485 , '0' , '0'  , -1 ,  NULL } };
21
22  /* IP.M Transition Table */
23  static INTEGER When1b4 [] = {473 , 0, 1, 2,  NULL };
24  ***
25  static INTEGER * Whenb004  /* ATP_ENTITY_BODY */  [] =
26   {When1b4, When2b4, When3b4, When4b4, When5b4, When6b4,
27    When7b4,When8b4, When9b4, When10b4, NULL};
28
29  /* STATE Transition Table */
30  static INTEGER From1b4 [] = {485, 1 , 2 , 3 , 7 , 8 , 15 ,
31     34 , 35 , 36 , 37 , 38 , 41 , 56 , 57 ,  NULL };
32  ***
33  static INTEGER * Fromb004  /* ATP_ENTITY_BODY */  [] = {
34     From1b4,   From2b4,   From3b4,   From4b4,   From5b4,
35     From6b4, From7b4,    NULL };
```

Figura 6.45 Tabela de Transição, Tabela When e Tabela From

- a tabela de transição **TrTab###** é criada (linhas 5 a 9, Figura 6.45);

- a tabela **InTab###** é criada para a inicialização do módulo (linhas 18 a 20, Figura 6.45);

- a tabela WHEN, composta pelas variáveis **When$b###** e **Whenb###**, é criada (linhas 22 a 27, Figura 6.45);

- a tabela FROM, composta pelas variáveis **From$b###** e **Fromb###**, é criada (linhas 29 a 35, Figura 6.45).

(h) função **Gub###**

A função **Gub###** é responsável pela seleção da transição a ser disparada. Para isto, ela utiliza as tabelas de transições e a função TrEvb### para obter a transição a ser disparada.

O comportamento dessa função pode ser resumido da seguinte forma (Figura 6.46):

- a lista de transições espontâneas disparáveis, **ListSpon**, é obtida (linha 18), através da comparação da lista de transições espontâneas (obtida na linha 17) com a lista de transições disparáveis no estado vigente (obtida na linha 15);

- a lista de transições não espontâneas disparáveis, **ListWhen**, é obtida (linha 26), através da comparação da lista de transições não espontâneas (obtida nas linhas 19 a 25) com a lista de transições disparáveis no estado vigente (obtida na linha 15);

- as listas **ListSpon** e **ListWhen** são concatenadas (linha 27) e ordenadas pela função **SortPrio** de acordo com a

prioridade de execução definida pela cláusula PRIORITY (linha 28);

- as transições são avaliadas (linhas 29 e 30), ativando-se a função TrEvb###.

```
01 Gub004  /* ATP_ENTITY_BODY */ (ctx_p)
02 _ctxctrl * ctx_p;
03
04 {Tb004  /* ATP_ENTITY_BODY */ * Int ;
05  Th004  /* ATP_ENTITY */ * Ext ;
06  Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
07  Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
08  /* TRANS-EVALUATION */
09  {INTEGER ListTrans  [59] ;
10   INTEGER ListWhen   [52] ;
11   INTEGER ListSpont  [7]
12   INTEGER inter           ;
13   INTEGER ip              ;
14   _ip   * ip_p            ;
15   GetListTrans(Fromb004,Int->MajorState,ListTrans);
16   *ListSpont = NULL ;
17   GetListSpont(TrTab004,ListSpont,57);
18   SortFrom(ListSpont,ListTrans);
19   *ListWhen = NULL;
20   ip_p = _whenfip(ctx_p);
21   while(ip_p)
22     {inter = _getinter(ctx_p,ip_p) ;
23      ip    = ip_p-> _SimIp.ei;
24      AppWhen(Whenb004,inter,ip,ListWhen);
25      ip_p  = _whennip(ctx_p,ip_p);};
26   SortFrom(ListWhen,ListTrans);
27   CatList(ListSpont,ListWhen,ListTrans);
28   SortPrio(ListTrans,TrTab004 );
29   EvalTrans(ctx_p,ListTrans,57,TrEvb004,Int->TabTrans,
30             TrTab004, NULL,NULL,NULL,NULL,NULL);
31  };
32  return(Int -> TabTrans[0].num);
33 } /*End of procedure definition*/ /*ATP_ENTITY_BODY*/
```

Figura 6.46 Código da função Gub### (seleção da transição)

(i) função **Inb###**

A função **Inb###** implementa as construções responsáveis pela parte de inicializações do corpo de um módulo (comando INITIALIZE). São ativadas as funções InEvb### (que implementa a cláusula PROVIDED) e InExb### (que implementa o bloco de comandos).

Após o teste de habilitação, realizado pela função InEvb### (linhas 9 a 21, Figura 6.47), a função InExb### é ativada e a parte de inicializações é executada (linhas 23 e 24, Figura 6.47).

```
01 Inb004  /* ATP_ENTITY_BODY */ (ctx_p)
02 _ctxctrl * ctx_p;
03 {Tb004  /* ATP_ENTITY_BODY */ * Int ;
04  Th004  /* ATP_ENTITY */ * Ext ;
05  Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
06  Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
07
08  /* INIT-EVALUATION */
09  {INTEGER nb_tt                    ;
10   INTEGER index                    ;
11   INTEGER ListTrans  [2]  ;
12   MakListTrans(1,ListTrans);
13   index = 0 ;
14   nb_tt = 0 ;
15   while(ListTrans[index])
16     {if (InEvb004(ctx_p,ListTrans[index]))
17        {nb_tt ++ ;
18         PutTT(&(Int -> TabInit[nb_tt]),ListTrans[index],
19               InTab004,NULL); } ;
20      index ++ ; };
21   Int -> TabInit[0].num = nb_tt ;
22   /* INIT-EXECUTION */
23   if (nb_tt) InExb004(ctx_p,
24      Int->TabInit[Randomize(nb_tt)].num);
25  };
26  return(Int -> TabInit[0].num);
27 } /* End of procedure definition */ /* ATP_ENTITY_BODY */
```

Figura 6.47 Código da função Inb### (inicialização do Corpo)

(j) função **Ihbb###**

A função **Ihbb###** (Figura 6.48) é responsável pela inicialização das estruturas do cabeçalho de um módulo, durante a criação de uma instância desse módulo.

```
01 /* Header and Body data Initialize Procedure */
02 Ihbb004  /* ATP_ENTITY_BODY */ (ctx_p)
03 _ctxctrl * ctx_p ;
04 {Tb004  /* ATP_ENTITY_BODY */ * Int ;
05  Th004  /* ATP_ENTITY */ * Ext ;
06  INTEGER i [50]    ; /* Control Variables */
07  INTEGER lowb [50] ; /* low bounds Variables */
08  Int  = (Tb004  /* ATP_ENTITY_BODY */  *) ctx_p ;
09  Ext  = (Th004  /* ATP_ENTITY */  *) ctx_p ;
10  ctx_p->_SimCtx._firstip_p  = NULL ;
11  ctx_p->_SimCtx.FirTrans    = Int -> TabTrans ;
12  ctx_p->_SimCtx.NFirTrans   = Int -> TchTrans ;
13  ctx_p->_SimCtx.InitTrans   = Int -> TabInit  ;
14  Int -> TabTrans [0] . num  =  NULL ;
15  Int -> TchTrans [0] = NuTrb004  /* ATP_ENTITY_BODY */ ;
16  (Int->TIMER1 )._SimIp._nextip_p=ctx_p->_SimCtx._firstip_p;
17  ctx_p->_SimCtx._firstip_p = &( Int->TIMER1 );
18  (Int->TIMER1 )._discipl = _individual;
19  (Int->TIMER1 )._SimIp.ei=483;
20  (Int->TIMER2)._SimIp._nextip_p=ctx_p->_SimCtx._firstip_p;
21  ctx_p->_SimCtx._firstip_p = &( Int->TIMER2 );
22  (Int->TIMER2 )._discipl = _individual;
23  (Int->TIMER2 )._SimIp.ei=484;
24  (Ext->TSAP)._SimIp._nextip_p= ctx_p->_SimCtx._firstip_p;
25  ctx_p->_SimCtx._firstip_p = &( Ext->TSAP );
26  (Ext->TSAP )._discipl = _individual;
27  (Ext->TSAP )._SimIp.ei=473;
28  (Ext->MAPX )._SimIp._nextip_p = ctx_p->_SimCtx._firstip_p;
29  ctx_p->_SimCtx._firstip_p = &( Ext->MAPX );
30  (Ext->MAPX )._discipl = _individual;
31  (Ext->MAPX )._SimIp.ei=474;
32 }
```

Figura 6.48 Código da função Ihbb004 (inicialização do cabeçalho)

As variáveis de controle, que implementam a lista de transições, são inicializadas (linhas 10 a 13). Em seguida, são inicializadas os pontos de interação do módulo (linhas 18 a 19, 22 a 23, 26 a 27 e 30 a 31), sendo criada a ligação das estruturas de dados, que formam os pontos de interação de um módulo (linhas 10, 16 a 17, 20 a 21, 24 a 25 e 28 a 29).

6.4.2.3. Geração da Tabela de funções e da Tabela de módulos

A tabela de funções executáveis (_bdesc) e a tabela de descrição de módulos (_hdesc) são geradas no final do código.

A tabela _bdesc contém as três funções básicas geradas para o corpo de cada módulo: Inb### (inicialização do corpo do módulo), Gub### (seleção da transição a ser disparada) e Trb### (execução da transição).

O vetor **_bdesc** (linhas 1 a 9, Figura 6.49) contém o endereço das funções descritas anteriormente. Isto permite ao núcleo de exploração selecionar o módulo a ser executado, sem conhecê-las prévia e explicitamente.

O vetor **_hdesc** (linhas 11 a 19, Figura 6.49) contém a tabela de descrição dos módulos, onde o atributo associado a cada módulo ("process" ou "activity") é descrito.

```
01  /* EXECUTION TABLE  */
02  _bdesctype _bdesc [ ]  = {
03   {Gub000,Trb000,Inb000}, /* TRANSPORT_IMP    */
04   {Gub001,Trb001,Inb001}, /* S_INT_BODY       */
05   {Gub002,Trb002,Inb002}, /* N_INT_BODY       */
06   {Gub003,Trb003,Inb003}, /* TRANSPORT_BODY   */
07   {Gub004,Trb004,Inb004}, /* ATP_ENTITY_BODY  */
08   {Gub005,Trb005,Inb005}, /* T_CLOCK_BODY     */
09   {Gub006,Trb006,Inb006}, /* MAPPING_BODY     */ }
10
11  /*  HEADERS DESCRIPTION TABLE  */
12  INTEGER _hdesc [ ]   = {
13   _frame      , /* TRANSPORT_IMP */
14   _process    , /* S_INTERFACE   */
15   _process    , /* N_INTERFACE   */
16   _process    , /* TRANSPORT     */
17   _process    , /* ATP_ENTITY    */
18   _process    , /* T_CLOCK       */
19   _process    , /* MAPPING       */ }
```

Figura 6.49 Tabela de processos e de execução

6.5. Complementação do Código

Ao código gerado deve ser acrescentado as rotinas de suporte fornecidas pelo núcleo de implementação **ESKIMO**. Essas rotinas desempenham as seguintes funções: gerência de buffers, núcleo de exploração e comunicação entre processos.

As rotinas para a gerência de buffers, extremamente úteis no auxílio às implementações dos protocolos referentes ao modelo OSI, são independentes do ambiente operacional, uma vez que a função padrão **calloc** é utilizada.

O núcleo de exploração dá suporte apenas às especificações que contenham um único módulo do tipo sistema (**systemprocess** ou **systemactivity**). Isto porque é necessário que cada módulo sistema corresponda a um processo no ambiente operacional, para que o paralelismo assíncrono, existente entre os módulos desse tipo, possa ser efetivamente implementado. Assim sendo, uma especificação deve ser desmembrada em tantas especificações quantos forem os seus módulos sistema.

A última função das rotinas do **ESKIMO** é permitir a comunicação entre os processos correspondentes aos módulos sistema da especificação original. O modelo adotado para essa comunicação baseia-se em caixas postais ("mailboxes"). Cada caixa postal tem um nome e somente o seu dono pode ler o conteúdo. Os demais processos podem somente mandar informações para essa caixa.

Para que as subrotinas de gerência de "buffer" e as subrotinas que implementam "mailboxes" possam ser utilizadas, estas devem ser declaradas na especificação como sendo subrotinas do tipo "primitive" (linhas 14 a 19, Figura 6.50). As estruturas de dados a serem manipuladas por essas subrotinas devem ser declaradas com o auxílio de "..." (linha 4, Figura 6.50). O comentário qualificador (**{$ generate}**) é utilizado para conectar

os tipos genéricos "..." e os tipos previamente definidos na linguagem C (linha 3, Figura 6.50).

```
01 type
02   (* tipos manipulados pelas funções de mailbox e buffer *)
03   {$ generate 'mxid'    'ecmtyp'}
04   My_mxid  = ...;
05   {$ generate 'mxdesc'  'ecmtyp'}
06   My_mxdesc= ...;
07   ***
08   {$ generate 'buff' 'buftyp' }
09   user_buffer = ...;
10   {$ generate 'char'
11   user_octet  = 0 .. 256;
12
13   ***
14   procedure mxiop (loc_mxid: My_mxid); primitive;
15   function  mxoop (rem_mxid: My_mxid): my_mxdesc;
16      primitive;
17   ***
18   function bufnul(b:user_buffer): boolean; primitive
19   procedure bufrst(var b:user_buffer); primitive
```

Figura 6.50 Declaração das funções de suporte do ESKIMO ("buffer" e "mailboxes")

Uma vez que essas subrotinas são declaradas, elas podem ser utilizadas em qualquer ponto da especificação.

Além das rotinas de suporte fornecidas por ESKIMO, outras rotinas podem ser criadas ou utilizadas. Essas rotinas devem ser tratada também como subrotinas do tipo "primitive" e as estruturas de dados específicas devem ser declaradas.

As subrotinas que implementam o núcleo de exploração ("scheduler/dispatcher") não precisam ser declaradas. O funcionamento do "scheduler/dispatcher" é simples:

- na função principal (MAIN.C), são ativadas as funções **_runtimer** (inicializa o relógio) e **_erun** (inicia a avaliação);

- a função **_erun** (DISPATCH.C) inicializa a especificação e entra num laço eterno;

- nesse laço, o temporizador é inicialmente avaliado ("_Vgettic"). Em seguida, uma instância de um módulo é selecionada para execução ("_eeval") e posteriormente é executada (através da ativação das funções descritas na tabela **_bdesc**). Pode-se inicializar uma instância (Inb###) ou avaliar suas transições (Gub###) para executá-las posteriormente (Trb###).

Outras rotinas de suporte podem ser também implementadas, visando complementar as facilidades oferecidas por **ESKIMO**. É importante atentar para a transportabilidade dessas rotinas, a fim de facilitar a sua adaptação a qualquer ambiente operacional.

Em relação ao código de implementação do TP2 (ou de qualquer outro protocolo) os únicos componentes dependentes do ambiente operacional são algumas rotinas de suporte do **ESKIMO**, que implementam as primitivas Estelle e as caixas postais (mailboxes), e as rotinas específicas para a comunicação com as camadas adjacentes.

As primitivas Estelle podem ser implementadas como chamadas ao supervisor do ambiente operacional. São rotinas responsáveis pela obtenção de informações de temporização (_ppinit, _vclock, _vget tic, _pnbdy, signal, alarm).

As rotinas que implementam as caixas postais (mxcontent, mxhread, mxiread, mxwrite, mxiop, mxoop, mxoclos e mxiclos) utilizam os mecanismos de comunicação entre processos, sendo, portanto, dependentes do suporte dado pelo ambiente operacional.

As rotinas de interface com as camadas adjacentes devem ser implementadas em função das próprias implementações dessas camadas. Por exemplo:

- se a camada adjacente é um processo do sistema operacional, as rotinas de interface devem utilizar as facilidades de comunicação entre processos;

- se a camada adjacente está implementada em "firmware", as rotinas da interface devem se comunicar com o "driver" desse dispositivo;

- se existe uma "Aplication Program Interface (API)", que dá acesso aos serviços da camada, as rotinas devem ser adaptadas a fim de utilizar as facilidades oferecidas;

- se a camada adjacente está no núcleo do sistema operacional, as rotinas de interface devem realizar chamadas ao supervisor desse sistema.

6.5.1. Arquitetura da Implementação do TP2

Como exemplo de implementação do TP2 para "workstation" Sun/3 (Figura 6.51), pode ser realizada a comunicação entre processos no sistema operacional UNIX através de "message-queues". A comunicação com "Internet Protocol" (camada de Rede) pode ser realizada utilizando o mecanismo "raw sockets". As informações de temporização podem ser obtidas através de alguma biblioteca provida pelo compilador C (por exemplo, signal.h).

Maiores detalhes sobre os mecanismos citados acima ("sockets" e "message-queues") podem ser obtidos em [Sun 90a] e [Sun 90b].

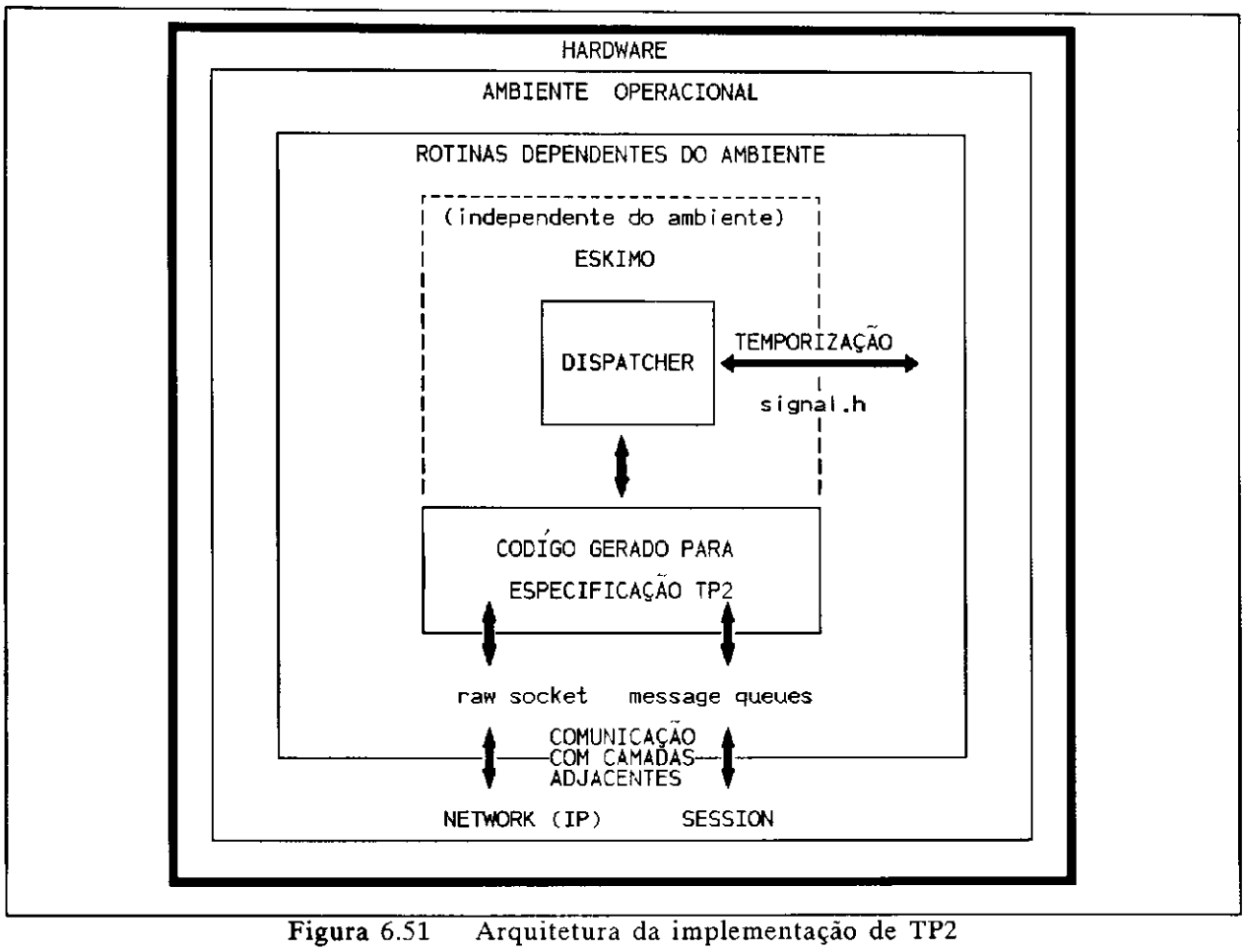

Figura 6.51 Arquitetura da implementação de TP2

6.6. Geração da Implementação Final

Decorridas as etapas precedentes, três grupos de códigos podem ser identificados:

- código gerado a partir da especificação formal;
- código referente às rotinas de suporte independentes do ambiente operacional;
- código referente às rotinas dependentes do ambiente operacional.

Os códigos pertencentes aos dois primeiros grupos são submetidos ao compilador C. Os códigos pertencentes ao terceiro

grupo devem ser submetidos a um compilador específico. Em seguida, os códigos objeto resultantes devem ser ligados às bibliotecas correspondentes, obtendo-se assim um código executável para um ambiente operacional específico. A arquitetura da implementação final é mostrada na Figura 6.52.

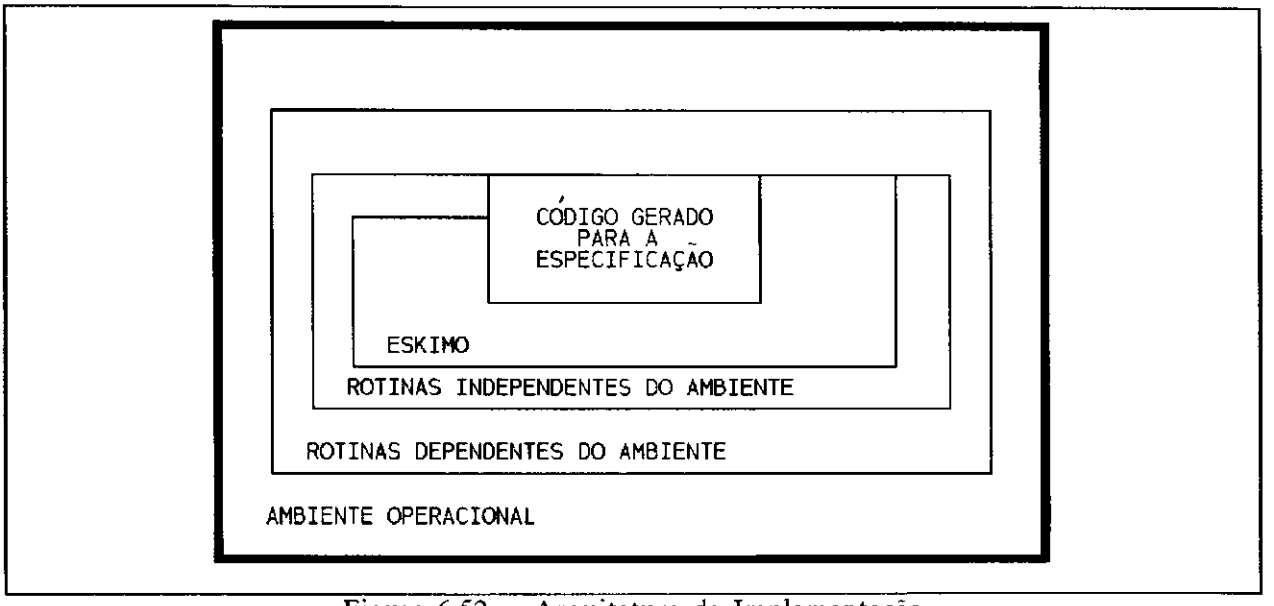

Figura 6.52 Arquitetura da Implementação

Em relação à Figura 6.52, o código gerado, a partir da especificação, utiliza tanto as facilidades oferecidas pelas rotinas de suporte independentes do ambiente operacional (**ESKIMO** e outras rotinas complementares), quanto as facilidades oferecidas pelas dependentes desse ambiente (e.g, rotinas para comunicação com as camadas adjacentes). **ESKIMO**, por sua vez, utiliza as facilidades oferecidas pelas rotinas dependentes do ambiente operacional, para obter informações de temporização e para se comunicar com outros processos desse ambiente.

7. Conclusão

Neste trabalho foi apresentada uma metodologia para derivação semi-automática de implementações de protocolo. As principais etapas dessa metodologia são: especificação formal do protocolo, validação da especificação, geração do código de implementação e complementação do código de implementação.

A especificação formal do protocolo de Transporte Classe 2 (TP2), realizada na TDF Estelle, foi gerada da seguinte forma:

- inicialmente, foi obtida uma especificação abstrata;
- em seguida, foi obtida uma especificação orientada para implementação, definindo-se completamente as suas estruturas de dados e as suas subrotinas.

A especificação foi validada, empregando-se a ferramenta SIMULATOR/DEBUGGER do ambiente EWS. Os traços obtidos na simulação do serviço foram comparados com os traços obtidos na simulação do protocolo, para validar o design do protocolo.

As interfaces com as camadas adjacentes foram adicionadas à especificação formal validada, antes da geração automática do código de implementação. A geração automática do código "garante" a sua conformidade em relação à especificação.

A complementação do código compreende a adaptação das rotinas de suporte oferecidas por EWS (ESKIMO) ao ambiente operacional e a implementação das rotinas que realizam as interfaces com as camadas adjacentes.

Finalmente, o código executável do protocolo foi gerado da seguinte forma:

- o código C gerado a partir da especificação Estelle e o código das rotinas de suporte são compilados no compilador C;

- as rotinas de interface são compilados no compilador específico;

- os códigos objetos resultantes são ligados às bibliotecas específicas, gerando o código executável do protocolo.

Para o protocolo TP2, foram geradas em torno de 14000 linhas de código C para 5000 linhas de especificação Estelle (Tabela 7.1). Considerando-se as rotinas de suporte e o código correspondente ao TP2 (Tabela 7.2), verifica-se que cerca de 95% do código C foi gerado automaticamente. Isso garante a transportabilidade e legibilidade desse código, uma vez que a codificação personalizada é eliminada.

Componente da Especificação	Nº LINHAS
ATP	1000
MAPPING	1000
NETWORK_INTERFACE	250
SESSION_INTERFACE	250
subrotina genéricas	1000
subr. codificação/decodificação	1500

Tabela 7.1 Características da Especificação

A metodologia proposta reduz drasticamente o tempo normalmente requerido para o desenvolvimento de uma implementação. Enquanto que Voung [VLC 88] consumiu um ano para implementar manualmente o TP2, a especificação Estelle (detalhada e validada) e a implementação do TP2, produzidas neste trabalho, consumiram dois meses. No último caso, não foi contabilizado o tempo gasto para obter conhecimentos sobre TP2, Estelle e EWS.

Uma comparação entre a implementação obtida neste trabalho e as implementações de Voung (manual e gerada pelo compilador

Estelle-C) é apresentada na Tabela 7.2. O tamanho das rotinas de suporte não foi mencionado em [VLC 88]. É importante ressaltar que EWS implementa a versão atual de Estelle e Estelle-C implementa a versão Estelle/84. Além disso, a especificação obtida neste trabalho implementa uma versão completa do TP2, contém diversas funções para tratamento de exceções e contém as funções de empacotamento/desempacotamento.

Componente da Implementação	(Nº LINHAS) EWS	(Nº LINHAS) Estelle_C	(Nº LINHAS) IMP. MANUAL
TP2	14000	6300	3500
rotinas de suporte	7000	(não disponível)	(não disponível)
rotinas de interface	1200 (valor estimado)	1250	1250

Tabela 7.2 Análise Comparativa dos códigos gerados

Em geral, implementações geradas automaticamente apresentam desempenho inferior em relação a uma implementação manual. Analisando o código gerado por EWS, é possível verificar que o desempenho pode ser melhorado com a retirada de rotinas de teste de consistência (divisão por zero, etc) e com a implementação, através de uma linguagem de baixo nível, das subrotinas mais utilizadas e mais complexas.

O código gerado por EWS pode ser utilizado para gerar um protótipo inicial, a partir do qual pode ser gerada uma implementação manual mais eficiente.

Embora o código gerado, segundo essa metodologia, seja relativamente extenso, dependendo dos requisitos do usuário, a minimização do esforço de desenvolvimento justifica a utilização de ferramentas automáticas em detrimento da performance obtida com a programação manual.

Estudos futuros, visando melhorar o desempenho e reduzir o tamanho do código gerado automaticamente, devem viabilizar a utilização em larga escala da metodologia proposta nesta dissertação.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARCG 84] Ansart, J.P., Rafiq, O., Castanet, R. e Guilton, P.: "Some operational tools in a OSI protocols study environment", Proceeding of SIGCOMM 84, Montreal, junho de 1984, p. 156-161.

[Balz 85] Balzer, R.: "A 15 year perspective on automatic programming", IEEE Trans. Soft. Enge., v. SE-11, Nº 11, novembro de 1985, p. 1257-1268

[BaPe 84] Basili, V.R. e Perricone, B.T.: "Software errors and complexity: an empirical investigation", communications on ACM, v. 27, nº 1, janeiro de 1984, p. 42-52.

[BGS 84] Bochmann, G.v., Geber, G., Serre, J.M.: "Semi-automatic implementation of communication protocols", publication #555, Universidade de Montreal, Canadá, janeiro de 1984, 18 p.

[BGS 86] Bochmann, G.v., Geber, G., Serre, J.M.: "A metodology for implementing high level protocols", publication #518, Universidade de Montreal, Canadá, janeiro de 1986, 14 p.

[BGS 87] Bochmann, G.v., Geber, G., Serre, J.M.: "Semiautomatic implementation of communication protocols", IEEE Tran. Soft. Enge., SE-13, Nº 9, setembro de 1987, p. 989-1000.

[Bock 83] Bockmann, G.v.: "Concepts of distributed systems design", Springer-Verlag, Berlin (Alemanha), 1983.

[CCIT 80] CCITT Recomendação S.70 - "Network-Independent Transport Service for Teletex", novembro de 1980.

[CCIT 84] CCITT Recomendação X.200 - "Reference model of open systems interconnection for CCITT applications", outubro de 1984.

[CCIT 88a] CCITT Recomendação X.214 - "Transport Service Definition", Livro Azul ("Blue Book"), 1988, v. VIII.4, p. 278-302

[CCIT 88b] CCITT Recomendação X.224 - "Transport Protocol Specification", Livro Azul ("Blue Book"), 1988, v. VIII.5, p. 36-128

[CCIT 88c] CCITT Recomendação X.213 - "Network Service Definition", Livro Azul ("Blue Book"), 1988, v. VIII.4, p. 219-278

[CCIT 88d] CCITT Recomendação X.223 - "Network Protocol Specification", Livro Azul ("Blue Book"), 1988, v. VIII.5, p. 6-35

[CCIT 88e] CCITT Recomendação X.215 - "Session Service Definition", Livro Azul ("Blue Book"), 1988, v. VIII.4, p. 303-385.

[CCIT 88f] CCITT Recomendação X.225 - "Session Protocol Specification", Livro Azul ("Blue Book"), 1988, v. VIII.5, p. 129-270.

[CCIT 88g] CCITT Recomendação X.121 - "International Numbering Plan for Public Data Network", Livro Azul ("Blue Book"), 1988, v. VIII.3, p. 317-342

[Dant 80] Danthine, A.A.S.: "Protocol representation with finite state models", IEEE Tran. Comm., v. COM-28, abril de 1980, p. 632-642.

[ECMA 81] Standard ECMA-72. Transport Protocol, janeiro 1981.

[EWS 89] EWS User's manuals, ESPRIT project 1265 - SEDOS Estelle Demonstrator, junho de 1989, 236 p.

[Fern 88] Ferneda, E.: "Um compilador para a técnica de descrição formal Estelle/83", dissertação de mestrado, UFPb - Campina Grande(PB), maio de 1988, 192 p.

[GoMo 91] Goldberg, S.H. e Mouton Jr., J.A.: "A base for portable communication software", IBM systems journal, v. 30, Nº 3, 1991, p. 259-279

[HoMu 84] Horowitz, E. e Munson, J.B.: "An expansive view of reusable software", IEEE Tran. Soft. Enge., v. SE-10, Nº 5, setembro de 1984, p. 477-487

[ISO 83a] ISO IS 7498: "Basic reference model for open systems interconnections", 1983

[ISO 83b] ISO IS 7185: "Programming Languages - PASCAL", 1983.

[ISO 86] ISO DIS 8072: "Transport Service Definition for open systems interconnections", Edição I, 1986, 22 p., JTC-1.

[ISO 88a] ISO IS 9074. ISO/TC 97/SC 21/20.1: "Open Systems Interconnection - Estelle - A formal description technique based on an extended state transition model", novembro de 1988, 179 p.

[ISO 88b] ISO DIS 8073: "Connection Oriented Transport Protocol Specification for open systems interconnections", Edição II, 1988, 115 p., JTC-1.

[KeRi 78] Kernighan, B.W. e Ritchie, D.M.: "The C Programming Language", New Jersey, Prentice-Hall, 1978.

[LiMa 83] Linn, R.J. e MaCoy, W.H.: "Producing tests for the implementation of OSI protocols", Testing OSI protocols: a compendium of papers, Relatório Nº ICST-SNA83-NBS, 1983, 18p.

[NBS 81] National Bureau of Standards: "A formal specification technique and implementation method for protocols", U.S. Department of Commerce, ICST/HLNP-81, julho de 1981, 46 p.

[PoSm 82] Pozefsky, D.P. e Smith, F.D.: "A meta-implementation for Systems Network Architecture", IEEE Trans. Comm., COM-30, junho de 1982, v.6, p. 1348-1355

[SaBo 82] Sarikaya, B. e Bochmann, G.v.: "Some experience with test sequence generation for protocols", Anais do 2º Int. Workshop on Protocol Specification, Testing and Verification, 1982.

[LoSt 88] Souza, W.L. e Stiubiener, S.: "Especificação, verificação e testes de protocolos", relatório técnico GRC/UFPb nº TR-01/88, fevereiro de 1988.

[Souz 90] Souza, J.N.: "Uma metodologia para validação, através de simulação, de especificações formais de protocolos de comunicação", dissertação de mestrado, UFPb - Campina Grande (PB), março de 1990

[Staa 83] Staa, A.v.: "Engenharia de Programas", LTC Editora, Rio de Janeiro-RJ, 1983, 286 p.

[Sun 90a] Network Programming Guide, Sun Microsystems Inc., EUA, 1990, 353 p.

[Sun 90b] Programmer's Overview Utilities & Libraries (System Services Overview / Programming Utilities & Libraries), Sun Microsystems Inc., 1990, 526 p.

[VLC 88] Vuong, S.T., Lau, A.C. e Chan, R.I.: "Semiautomatic implementation of protocols using an Estelle-C compiler", IEEE Trans. Soft. Enge., v. 14, Nº 3, março de 1988, p. 384-393.

[WaGu 82] Wasserman, A.I. e Gu*/tz, S.: "The future of programming", communication of ACM, v. 25, nº 3, março de 1982, p. 196-206.