IMPLEMENTAÇÃO SEMI-AUTOMÁTICA DE SISTEMAS DE COMUNICAÇÃO ESPECIFICADOS FORMALMENTE (*)

Wanderley Lopes de Sousa (**), José Neuman de Sousa

GRC/DSC/CCT/Universidade Federal da Paraíba

Av. Aprígio Veloso, 882
58100 Campina Grande (Pb)

Resumo - Este trabalho demonstra o emprego do Compilador Estelle/83, desenvolvido junto ao GRC/UFPb, na implementação semi-automática de sistemas de comunicação especificados em Estelle/83. Após uma breve introdução a essa linguagem e ao compilador, é dado a ênfase às estruturas de dados, referentes às construções próprias de Estelle/83, criadas pelo compilador. Em seguida é realizada uma análise de um segmento do código gerado pelo compilador, a partir de uma especificação Estelle/83 do protocolo Abracadabra (definido pelo CCITT). Finalmente são apresentadas as conclusões e sugeridas algumas aplicações, envolvendo diferentes tipos de sistemas de comunicação.

Abstract - In this paper we demonstrate the use of the Estelle/83 Compiler, developed by GRC/UFPb, for the semiautomatic implementation of communication systems. After a brief introduction to the Estelle/83 language and after a brief description of this compiler, we explain the data structures created by the compiler. We also analyse a code segment generated by the compiler from a Estelle/83 specification of the Abracadabra protocol (defined by CCITT). Finally, we draw our conclusions and we suggest some applications for different types of communication systems.

1. INTRODUÇÃO

Num sistema de comunicação, onde as interações entre as entidades comunicantes são realizadas através da troca de mensagens, denomina-se **protocolo de comunicação** ao conjunto de regras que garante uma troca ordenada dessas mensagens. A descrição dessas regras é chamada de **especificação do protocolo.**

A fase crítica no ciclo de vida de um sistema de comunicação é a construção de uma especificação precisa de seus componentes. O uso de linguagens naturais, para a descrição de sistemas complexos, produz especificações informais geralmente ambíguas e inconsistentes. Para evitar tais problemas, Técnicas de Descrição Formal (TDFs) têm sido utilizadas na especificação de serviços e de protocolos de comunicação. Essas especificações formais simplificam também os esforços de validação, implementação e teste dos protocolos.

Órgãos internacionais de padronização, tais como o National Bureau of Standards (NBS), o Comité Consultatif International Télégraphique et Téléphonique (CCITT) e a International Organization for Standardization (ISO), têm desenvolvido TDFs para a especificação formal dos serviços e protocolos relativos ao modelo de referência Open Systems Interconnection (OSI). (ISO IS 7498)

Este trabalho ilustra o processo de obtenção semi-automática da implementação de um protocolo, a partir de sua especificação na TDF Extended State Transition Language (Estelle) (ISO IS 9074) versão de 1983, empregando-se o compilador desenvolvido junto ao Grupo de Redes de Computadores (GRC) da UFPb.

2. A LINGUAGEM ESTELLE/83

Estelle/83 (ISO TC97) (uma das primeiras versões de Estelle) é baseada numa Máquina de Estados Finita Estendida (MEFE), que combina dois tipos de notações (fig. 2.1).

Em Estelle/83 uma especificação é constituída de um módulo (module) ou de um conjunto de módulos. Cada módulo é representado por uma caixa preta com portas

(*) realizado com o auxílio do CNPq e da CAPES faz parte de um programa de trabalho entre o GRC/UFPb e o Centro Científico Rio da IBM Brasil

(**) atualmente, Professor Convidado junto ao DT/FEE/UNICAMP - Caixa Postal 6101 - CEP 13081 - Campinas (SP)

	ESTADOS		VARIÁVEIS		
Estelle/83 = MEF	INTERAÇÕES	+	PARÂMETROS	=	MEFE
	TRANSIÇÕES		PRIORIDADES		
			Pascal [4]		

Figura 2.1. MEFE de Estelle/83.

```
channel Ident_Canal (Papel1, Papel2);
   by Papel1 : Ident_Interação (Lista_Parâmetros);
               Ident_Interação (Lista_Parâmetros);
   by Papel2 : Ident_Interação (Lista_Parâmetros);
               Ident_Interação (Lista_Parâmetros);
end  Ident_Canal
```

Figura 2.2. Sintaxe para a definição de um canal.

(port) de entrada/saída. Canais (channel) de comunicação bi-direcional podem interligar módulos através de suas portas.

Módulos trocam interações. Um módulo pode enviar uma interação (por uma de suas portas) a um outro módulo, desde que ambos estejam interligados através do mesmo canal. O canal é uma construção que permite desvincular a especificação das interações da especificação dos módulos (Fig. 2.2).

Essa especificação indica que os módulos, a serem conectados (através de suas portas) às extremidades desse canal, desempenharão as funções **Papel1** e **Papel2**. O que desempenhar a função **Papel1** pode emitir as primitivas definidas em **Papel1** e deve receber as primitivas definidas em **Papel2**. O contrário deve ocorrer com o módulo que desempenhar a função **Papel2**.

A comunicação entre módulos pode ser efetuada através de "rendez-vous" ou através de filas FIFO ("First In - First Out") associadas às portas desses módulos. No primeiro caso, para que dois módulos possam trocar interações é necessário que ambos (emissor e receptor) estejam prontos (comunicação síncrona). No segundo caso, a primitiva emitida é colocada na fila associada à porta do módulo receptor (comunicação assíncrona).

A especificação de um módulo é constituída de duas partes: cabeçalho e corpo. O cabeçalho de um módulo representa o seu nível mais alto de abstração e sua definição é baseada na descrição de suas portas. Esse cabeçalho define um tipo módulo. As variáveis de um tipo módulo representam cópias desse tipo e todas possuem a mesma visibilidade externa (Fig. 2.3).

```
module Ident_Cabeçalho;
   Ident_Porta   :   Ident_Canal (Papel),
          |
   Ident_Porta   :   Ident_Canal (Papel),
end Ident_Cabeçalho,
```

Figura 2.3. Sintaxe para a definição do cabeçalho de um módulo.

Nessa especificação são definidas (indiretamente) as interações de um módulo, vinculando-se suas portas aos canais. Uma porta do tipo **Ident_Canal (Papel)** troca as primitivas definidas no canal **Ident_Canal** e desempenha a função **Papel** em relação a esse canal (o que indica a direção das interações).

A especificação do corpo de um módulo pode ser realizada através do seu refinamento (**refinement**) em submódulos (Fig. 2.4) ou através de um processo (**process**) que define o seu comportamento (Fig. 2.5).

```
refinement Ident_Corpo for Ident_Cabeçalho;
   <refinamento do corpo>
   <"instanciação">
   <estabelecimento dos elos de comunicação>
end Ident_Corpo;
```

Figura 2.4. Sintaxe para o refinamento de um módulo.

```
process Ident_Corpo for Ident_Cabeçalho;
   <declarações>
   <"inicializações">
   <transições>
end Ident_Corpo,
```

Figura 2.5. Sintaxe para a definição de um processo

No refinamento de um módulo são especificados os seus submódulos (que por sua vez podem ser refinados) e os canais internos (caso haja comunicação entre esses submódulos). Terminado o refinamento, cópias dos submódulos filhos (variáveis do tipo módulo) são criadas e instanciadas (**init**) com os seus respectivos corpos (previamente definidos). Finalmente os elos de comunicação são estabelecidos: as portas das instancias dos submódulos, interligadas através de canais internos, são conectadas (**connect**) e as portas do módulo pai, que devem ser vinculadas às portas de seus filhos, são substituídas (**replace**) pelas portas das respectivas instâncias.

Na especificação de um processo são declarados os objetos a serem manipulados, tais como: as portas do módulo que possuem filas associadas (**queued**), constan-

```
trans
                                    {condições}
   priority     < número C N >                  {em relação às outras transições}
   when         < port evento >                 {interação de entrada}
   provided     < predicado >                   {expressão booleana habilitadora}
   from         < estado_a >                    {estado de controle vigente}
                                    {ações}
to              < estado_b >                    {próximo estado de controle}
   begin                                        {atualização das variáveis adicionais (Pascal)}
      .....
      .....
      out       < port evento >                 {interação de saída}
      .....
      .....
   end;
```

Figura 2.6. Sintaxe para a definição de uma transição

tes, tipos, variáveis, os estados de controle da MEFE (variável **state**), funções e procedimentos. Em seguida, são inicializadas a variável de estado de controle (também chamada de principal) e as variáveis de estado adicionais (Pascal). Finalmente, o comportamento do módulo é descrito através de um conjunto de transições (Fig. 2.6).

Cada transição é composta de duas partes: condições e ações. As condições são constituídas de cláusulas próprias a Estelle/83, sendo que a ordem dessas cláusulas é irrelevante e pelo menos uma cláusula deve ser declarada para cada transição. As ações são constituídas da cláusula **to**, de declarações Pascal, de extensões (por exemplo, **out**) e de restrições (e.g., os apontadores só podem ser utilizados na parte Pascal). Transições que não possuem a cláusula **when** são denominadas espontâneas.

A linguagem Estelle começou a ser desenvolvida em 1981, tornou-se um padrão da ISO em 24 de novembro de 1988 e permanecerá estável até 1993. Em relação a Estelle/83, as principais diferenças são relativas aos aspectos dinâmicos, que começaram a ser incorporados à linguagem a partir de 1985.

Em Estelle/83 a arquitetura de um sistema é criada estaticamente, ou seja, uma vez estabelecida a sua estrutura hierárquica em termos de módulos e submódulos e uma vez estabelecido os elos de comunicação, essa arquitetura não poderá mais ser modificada. No padrão atual é possível, além dessa definição estática, a criação e a destruição dinâmica de módulos, assim como o estabelecimento dinâmico dos elos de comunicação.

3. O COMPILADOR ESTELLE/83 (Ferneda-1988)

Esse compilador, conjuntamente com o Compilador UdeM (Gerber-1983), o Simulador Veda (Jard, Monin, Groz - 1986), o Compilador NBS e o Compilador UBC (Vuong, Lau, Chan - 1986), foi uma das primeiras ferramentas desenvolvidas para a linguagem Estelle. Os trabalhos foram iniciados em 1985, sendo que o documento (de definição da linguagem) utilizado na época foi o Working Document (WD) FDT B (ISO - 1983), de circulação interna ao grupo ISO TO 97/SC16/WG1, que iniciou o desenvolvimento de Estelle. O primeiro documento oficial da ISO foi o Draft Proposal DP 9074 de 04 de junho de 1986.

O Compilador Estelle/83 permite que uma especificação seja fragmentada em vários módulos fonte, podendo cada módulo ser compilado separadamente. Para cada fonte é produzido um segmento de código Pascal, sendo que o código objeto é gerado pelo compilador Pascal. Ao conjunto de códigos objeto, referentes aos fontes Estelle/83, devem ser aglutinados os códigos objeto relativos a uma biblioteca (rotinas de suporte e núcleo), o que produzirá um programa executável.

A compilação completa da especificação compreende as seguintes fases:

(a) análises léxica, sintática e semântica estática;

(b) geração de código.

O código Pascal, gerado pelo compilador a partir de uma especificação Estelle/83, pode ser dividido em:

(a) declarações: constantes e tipos próprios à especificação; registros que representarão os processos, portas (tipo canal) e interações; variáveis para o armazenamento dos endereços do processo, porta e interação que estão sendo tratados; cabeçalhos das funções e procedimentos a serem importados (externos);

(b) procedimentos relativos aos processos contidos na especificação;

(c) procedimentos que inicializam as estruturas de dados criadas pelo compilador.

A estrutura de dados referente a um processo, apresentada na Fig. 3.1, é composta de:

(a) o campo **IDENT** (do tipo sequência de caracteres) identifica o nome do processo;

(b) o campo **CHANLIST** (do tipo ponteiro) guarda o endereço da estrutura de dados (do tipo canal) da primeira porta a ser investigada, quando da execução do processo;

(c) o campo **NEXT** (do tipo ponteiro) guarda o endereço do próximo processo a ser investigado. Dessa forma, uma lista encadeada circular interliga todos os processos correspondentes às instâncias dos módulos;

(d) a **PARTE VARIANTE** armazena as variáveis locais ao processo.

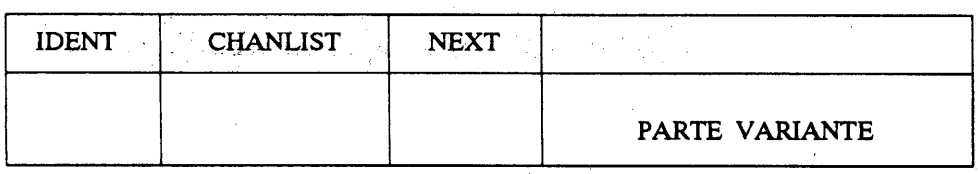

Figura 3.1. Estrutura de dados de um processo.

O compilador gera um tipo **PROCESS = (P0ident_processo, P0ident_processo, ...)**, onde são enumerados todos os processos especificados. A variável **P0VAR** guarda o endereço do processo que está sendo executado. A declaração da estrutura de um processo, em Pascal, é apresentada na Fig. 3.2.

```
01    TYPE
02      P1TYPE =  — > P0TYPE;
03      P0TYPE = RECORD
04                 IDENT  :  P2TYPE;
05                 CHANLIST  :  C1TYPE;
06                 NEXT : P1TYPE;
07                 CASE  PROCESS  OF
08                       P0ident_processo
09                         ( );
10                       P0ident_process
11                         ( );
12                       ...
13                 END;
```

Figura 3.2. Código Pascal relativo à estrutura de um processo.

IDENT	INDLIST	TARGET		NEXT	QUEUED	HEAD
		PROC	CHAN			

Figura 3.3. Estrutura de dados de uma porta.

A estrutura de dados de uma porta do tipo canal, apresentada na Fig. 3.3, é composta de:

(a) o campo **IDENT** (do tipo inteiro) identifica uma porta. Caso o processo contenha transições espontâneas, uma pseudo-porta (com **IDENT = 0**), criada pelo compilador, será a primeira a ser investigada;

(b) o campo **INDLIST** (do tipo ponteiro) percorre a lista de portas que são especificadas através do tipo **array**;

(c) o campo **TARGET** (do tipo registro) identifica o processo (**PROC**) e a porta (**CHAN**), ambos do tipo ponteiro, conectados à outra extremidade do canal;

(d) o campo **NEXT** (do tipo ponteiro) guarda o endereço da próxima porta a ser investigada. Dessa forma, uma lista encadeada permite percorrer todas as portas de uma mesma instância;

(e) o campo **QUEUED** (do tipo booleano) indica se há (**true**) ou não (**false**) um fila associada à porta. No caso negativo não existe o campo **HEAD**. Caso contrário, esse campo (do tipo ponteiro) guarda o endereço do último elemento colocado na fila.

O compilador gera um tipo **CHANNEL = (Ciident_canal, ..., CNident_canal, C0ident_processo,...)**, onde são enumerados todos os canais especificados. A variável **C0VAR** guarda o endereço da porta (do tipo canal) que está sendo tratada. A declaração da estrutura de uma porta, em Pascal, é apresentada na Fig. 3.4.

A estrutura de dados de uma interação, apresentada na Fig. 3.5, é composta de:

(a) o campo **NEXT** (do tipo ponteiro) é utilizado, no caso de comunicação através de filas, para construir uma lista encadeada de interações;

(b) na **PARTE VARIANTE** é declarado o corpo da interação. No caso de uma transição espontânea, uma pseudo-interação de entrada é declarada.

A variável **SOVAR** guarda o endereço da interação que está sendo tratada. A declaração em Pascal das possíveis interações, inclusive das pseudo-interações, é apresentada na (Fig. 3.6):

```
01    TYPE
02      P1TYPE   =  — > P0TYPE;
03      C1TYPE   =  — > C0TYPE;
04      S1TYPE   =  — > S0TYPE;
05      I1TYPE   =  — > I0TYPE;
06      I0TYPE   =     RECORD
07                       NEXT  :  I1TYPE;
08                       IDENT  : BOOLEAN
09                     END;
10      A0TYPE   =     RECORD
11                       PROC  :  P1TYPE;
12                       CHAN : C1TYPE
13                     END;
14      C0TYPE   =     RECORD
15                       IDENT  : INTEGER;
16                       INDLIST : I1TYPE;
17                       TARGET  : A0TYPE;
18                       NEXT  :  C1TYPE;
19                       CASE QUEUED  :  BOOLEAN OF
20                              FALSE :
21                                ( );
22                              TRUE  :
23                                (HEAD  ;  S1TYPE)
24                     END;
```

Figura 3.4. Código Pascal relativo à estrutura de uma porta.

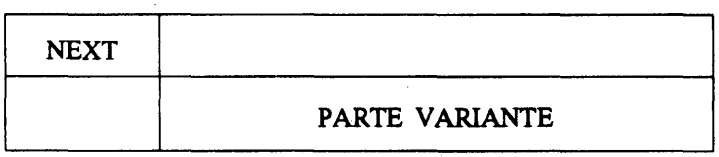

Figura 3.5. Estrutura de dados de uma interação.

```
S1ident_canal    =  (S1ident_interação, S1ident_interação, ...);
S2ident_canal    =  (S2ident_interação, S2ident_interação, ...);
...;
SNident_canal    =  SNident_interação, SNident_interação, ...);
S0ident_processo    =  (R1ANY);
S0ident_processo    =  (R2ANY);
...;
S0ident_processo    =  (RNANY);
```

Figura 3.6. Declaração em Pascal das interações.

Os tipos **S1ident_canal**, **S2ident_Canal**, ..., **SNident_canal** enumeram as interações declaradas na especificação dos respectivos canais. Os tipos **S0ident_processo** enumeram as pseudo-interações (**R1ANY**, **R2ANY**, ...) relativas às transições espontâneas dos respectivos processos. A declaração em Pascal da estrutura de uma interação é apresentada na Fig. 3.7.

```
01    TYPE
02      S0TYOE  =      RECORD
03                       CASE CHANNEL OF
04                         C1ident_canal
05                           ( );
06                         C2ident_canal
07                           ( );
08                         ... ;
09                         CNident_canal
10                           ( );
11                         C0ident_processo
12                           ( )
13                     END;
```

Figura 3.7. Código Pascal relativo à estrutura de uma interação.

As cláusulas Estelle/83, relacionadas à especificação das transições de um processo, são traduzidas da seguinte forma:

(a) **when** corresponde a duas declarações **if**. A primeira verifica se o identificador da porta vigente é o mesmo identificador especificado na cláusula. A segunda verifica se a interação, que está sendo tratada, é a mesma que consta na cláusula. Caso a interação possua parâmetros, o comando **with** é usado para permitir o acesso aos mesmos;

(b) **from** corresponde a uma instrução **if**, que verifica se o estado principal vigente (**state**) é o mesmo da cláusula;

(c) **to** corresponde à atribuição do novo estado principal, especificado na cláusula, à variável **state**;

(d) **provided** corresponde a uma declaração **if**, acompanhada da mesma expressão booleana presente na cláusula;

A instrução **out** é implementada da seguinte forma:

(a) é chamada a **P0PROCEDURE**, que faz parte das rotinas de suporte, para localizar a porta a ser tratada;

(b) é criada e inicializada a estrutura de dados da interação a ser emitida;

(c) é chamado o procedimento **OUT**, que faz parte do núcleo, para emitir a interação na porta adequada.

4. IMPLEMENTAÇÃO SEMI-AUTOMÁTICA DO PROTOCOLO ABRACADABRA

As especificações informal e formal do protocolo **Abracadabra** estão apresentadas em CCITT (1988). Esse documento é o resultado de um trabalho conjunto entre especialistas do CCITT e da ISO, cujo objetivo principal foi o de promover o emprego de TDFs na especificação dos padrões relativos ao modelo OSI.

O **Protocolo Abracadabra** é simétrico, opera entre um par de estações através de um meio "full-duplex" não confiável, fornecendo uma comunicação bilateral, simultânea e confiável ao par de usuários. As duas estações trocam as Unidades de Dados de Protocolo (UDPs) apresentadas na Fig. 4.1.

O Protocolo Abracadabra é parametrizado por duas constantes:

(a) **N** (> 0) define o número máximo de tentativas de transmissão de uma mesma UDP, sem a recepção do seu reconhecimento;

(b) P (> atraso total de trânsito da informação) define o tempo de espera para a retransmissão.

Em relação à fase de conexão:

(a) a sequência normal de PSs e UDPs, entre duas estações A (iniciadora) e B (respondedora) é **ConReq**{A}, **CR**{A--> B}, **ConInd**{B}, **ConResp**{B}, **CC**{B--> A}, **Conconf** {A};

(b) se for recebido **DisReq** ou **DR**, o protocolo entrará na fase de desconexão;

(c) outra **UDP** diferente de **CR**, **CC** ou **DR** deverá ser ignorada;

(d) se não houver resposta ao **CR** no intervalo P, este será retransmitido;

(e) após N tentativas sem sucesso, o protocolo entrará na fase de erro.

Em relação à fase de transferência de dados:

(a) a sequência normal de PSs e UDPs, entre duas estações A (emissoras) e B (receptora) é **DatReq**{A}, **DT**{A -- > B}, **DatInd**{B}, **AK**{B -- > A;

UDP	SIGNIFICADO	PRIMITIVAS (PS) CORRESPONDENTES	FASE
CR	requisição de conexão	ConReq, ConInd	Conexão
CC	confirmação de conexão	ConResp, ConConf	
DT	transferência de dados	DatReq, DatInd	Dados
AK	reconhecimento	----------	
DR	requisição de desconexão	DisReq, DisInd	Desconexão
CD	confirmação de desconexão	----------	ou Erro

Obs: somente as UDPS DT e AK possuem parâmetros. DT carrega uma PS e ambas carregam um bit de sequenciamento;

cada PS do Serviço Abracadabra é carregada numa única UDP DT e cada UDP do Protocolo Abracadabra é carregada numa única PS do Meio de Comunicação.

Figura 4.1. UDPs e correspondentes Primitivas de Serviço (PSs).

(b) se após a transmissão de um **DT** o **AK** correspondente não for recebido no intervalo **P**, o **DT** será retransmitido;

(c) após **N** tentativas sem sucesso, o protocolo entrará na fase de erro;

(d) **DTs** e **AKs** carregam um bit de sequenciamento, que é independente para cada direção de transmissão;

(e) esse bit é inicializado a 0, durante a conexão, e uma vez enviado um **DT**, o correspondente **AK** deverá carregar um bit cujo valor é diferente (o próximo) do transportado pelo **DT**;

(f) se o valor do bit transportado pelo **AK** for incorreto, o protocolo entrará na fase de erro;

(g) ao receber um **DT**, na **sequência correta**, um **DatInd** deverá ser emitido, assim como um **AK** contendo o próximo valor do bit. Caso **contrário**, somente um **AK** contendo o mesmo valor do bit recebido, deverá ser emitido;

(h) se dois **DT** forem recebidos, antes do envio do **AK** relativo ao primeiro **DT**, o protocolo entrará na fase de erro;

(i) ao receber um **DR** o protocolo entrará na fase de desconexão;

(j) se um **CR** adicional for recebido antes de qualquer **DT** ou **AK**, um **CC** deverá ser enviado;

(k) se outra **PDU**, diferente de **DT**, **AK**, **DR** ou **CR** (satisfazendo a condição anterior), for recebida, o protocolo entrará na fase de erro.

Em relação às fases de desconexão e erro:

(a) a sequência normal de PSs e UDPs, supondo a estação A iniciadora, é **DisReq** {A}, **DR** {A -- > B}, **DisInd** {B}, **DC** {B -- > A};

(b) se após a transmissão de um **DR** o **DC** não for recebido no intervalo **P**, o **DR** será retransmitido;

(c) após **N** tentativas sem sucesso, a conexão será considerada terminada;

(d) se após o envio do **DR** for recebido um **DR** ao invés do **DC** (**DisReq simultâneos**), a conexão será considerada terminada;

(e) se **DRs** adicionais forem recebidos, os **DCs** deverão ser emitidos;

(f) terminada a conexão, qualquer UDP diferente de **DR** ou **CR** deverá ser ignorada;

(g) ao detectar um erro no protocolo, a entidade deverá emitir um **DisInd** e um **DR**, nesta sequência. O protocolo entrará na fase de erro, que será idêntica à fase de desconexão.

Em relação ao meio de comunicação:

(a) é full-duplex e transparente, mas não garante a entrega de mensagens. Estas podem ser perdidas, mas não podem ser desordenadas, corrompidas, inventadas ou duplicadas;

(b) as primitivas **UnitReq** e **UnitInd** carregam as **UDPs** correspondentes ao Protocolo Abracadabra.

A arquitetura utilizada para a especificação formal Estelle/83 do Protocolo Abracadabra é apresentada na Fig. 4.2. O módulo **ABRA** é refinado em dois submódulos, sendo que **STATION** representa o protocolo propriamente dito. **TRANSCODE** empacota e desempacota as

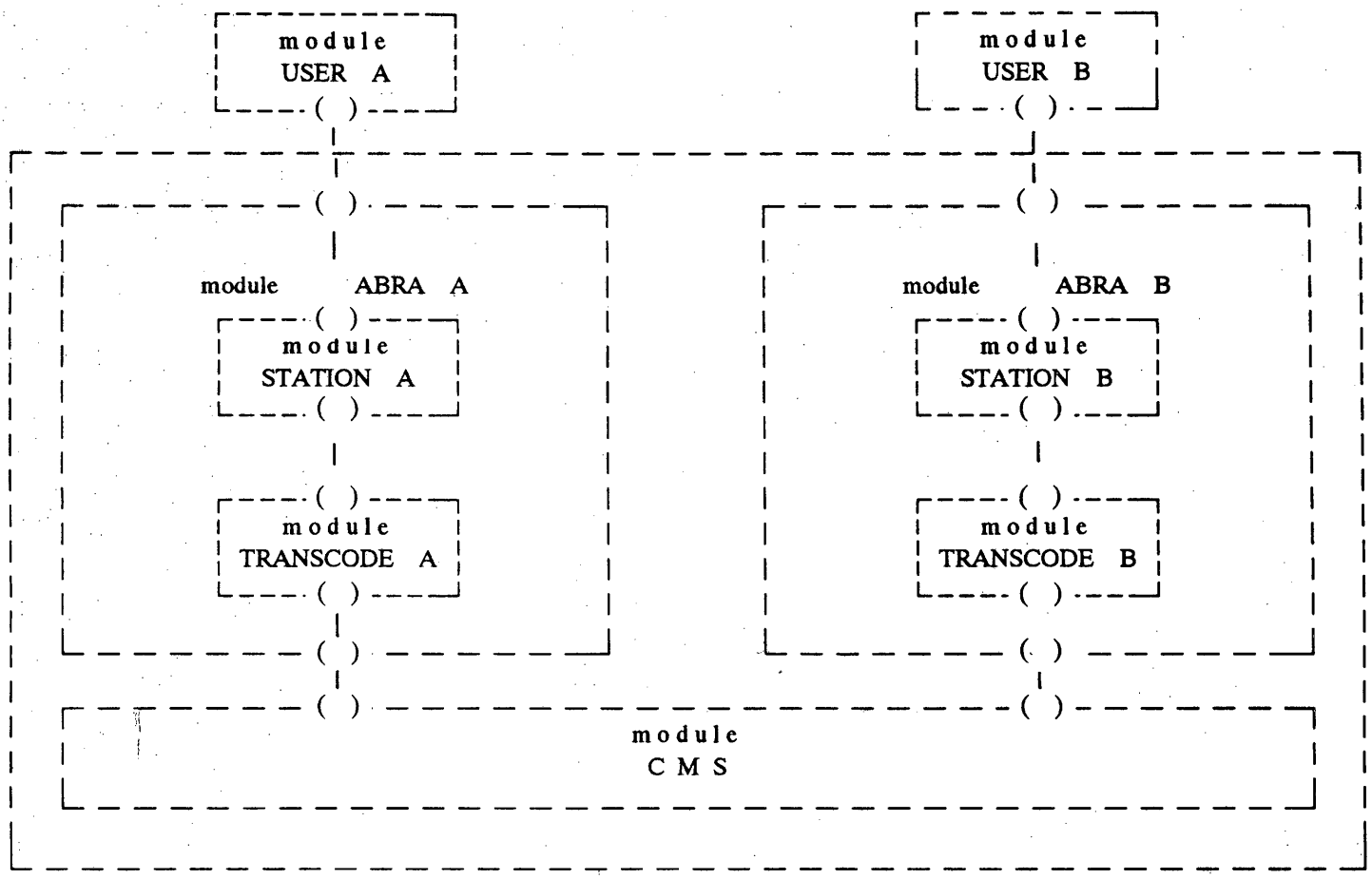

Figura 4.2 Arquitetura do Protocolo Abracadabra em Estelle/83.

UDPs emitidas e recebidas por **STATION**. Sugere-se ao leitor interessado na especificação completa que consulte (Souza, 90).

Um trecho da especificação do protocolo Abracadabra, submetida ao compilador, é apresentado no Anexo 1. O código Pascal correspondente é apresentado no Anexo 2.

As variáveis **P0VAR, C0VAR** e **S0VAR** são globais e guardam, respectivamente, os endereços das estruturas de dados dos processos, portas e interações vigentes. Se o procedimento **out** for executado **S0VAR** e **C0VAR** guardarão, momentaneamente, os endereços da interação emitida e da porta do processo receptor. Terminada a execução de **out**, os endereços originais deverão ser restabelecidos. A função de **S1VAR** e **C1VAR** é justamente guardar esses endereços.

A partir da linha 83 estão declaradas as **I0procedures**, que utilizam as rotinas de suporte para a manipulação das estruturas de dados. Para cada refinamento e para cada processo existe uma correspondente **I0procedure**. Por exemplo, **I0ABRA_BODY** (linhas 83-122) corresponde ao primeiro refinamento feito na especificação. Esse procedimento, ativado pelo núcleo, cria e interliga todas as estruturas de dados relativas ao protocolo especificado.

As declarações de **I0TRANSCODE_PROC** (linhas 101-108), ativado (linha 112) por **I0ABRA_BODY**, possuem as seguintes funções:

(a) **new (P0VAR, P0TRANSCODE_PROC)** atribui à variável **P0VAR** o endereço da estrutura de dados relativa ao processo **TRANSCODE_PROC**;

(b) **P1PROCEDURE** inicializa os campos da estrutura de dados relativa a esse processo;

(c) **P3PROCEDURE** liga a porta **1** (**Up** na especificação) à estrutura de dados relativa ao processo (campo **CHANLIST** de **TRANSCODE_PROC**);

(d) a segunda **P3PROCEDURE** liga a porta **2** (**Down**) à porta 1 (campo **NEXT** da estrutura de dados relativa à porta 1).

Em seguida **I0ABRA_BODY** ativa **I0STATION_PROC**, que desempenha funções semelhantes ao **I0TRANSCODE_PROC**. Terminada a execução desses procedimentos, as estruturas de dados relativas a cada processo (contido no refinamento **ABRA_BODY**)

estarão ligadas às estruturas de dados de suas respectivas portas. Em seguida é realizada a interconexão entre as estruturas de dados desses processos (linhas 114-121).

5. CONCLUSÃO

Os resultados obtidos com essa experiência específica (e que talvez possam ser generalizados) demonstraram que:

(a) um dos grandes problemas no desenvolvimento de uma implementação é a construção dos diferentes tipos de estruturas de dados e, principalmente, a interconexão dessas estruturas. O compilador faz isso automaticamente;

(b) o núcleo da implementação Pascal obtido é legível, não sendo necessário ao usuário, que conhece Estelle/83, muito esforço para visualizar o reflexo da especificação na implementação. O compilador elimina a codificação personalizada;

(c) esse núcleo, embora não seja otimizado (em relação a uma implementação manual), pode ser facilmente transportado;

(d) ele pode ser também utilizado como uma referência confiável para a análise do comportamento de uma implementação manual, desde que ambos tenham sido derivados da mesma especificação.

Foi desenvolvido, junto ao GRC/UFPb, o Simulador Estelle/83 (Jard, Monin, Groz - 1986). Ele é constituído basicamente de um núcleo e de um condutor de simulação, que aglutinados ao Compilador Estelle/83, permitem a validação, através de simulação, de especificações Estelle/83. Atualmente essas duas ferramentas estão sendo empregadas na especificação, na validação, e na implementação semi-automática do Protocolo de Transporte - Classe 2 e da função BCS do Protocolo de Sessão.

Estão sendo patrocinados, pela comunidade européia, os projetos European Strategic Programme for Research and Development in Information Technology (ESPRIT). Dentre os projetos concluídos em 1989, destaca-se o Software Environment for the Design of Open Distributed Systems (SEDOS)/Estelle Demonstrator (Nº 1265). Um dos objetivos desse projeto foi a investigação de possíveis áreas de aplicação da TDF Estelle. Algumas das áreas investigadas com sucesso foram: telecomunicações (sistemas de chaveamento ISDN), comunicação de dados (via satélite) e sistemas de controle industrial (células de robô).

A nossa experiência e principalmente os resultados obtidos no SEDOS/Estelle Demonstrador indicam que Estelle, embora tenha sido projetada visando a descrição formal dos protocolos das redes de computadores, pode ser empregada (e conseqüentemente as ferramentas a ela relacionadas) na especificação de outros tipos de sistema (distribuído ou não), desde que sua arquitetura possa ser descrita com as construções dessa TDF.

6. REFERÊNCIAS

CCITT COM X-R 29-E (1988), "Guidelines for the Application of Estelle, LOTOS and SDL", pp. 135-161.

JARD, C.; MONIN, J.F.; GROZ, R. (1986), "Experience in implementing X.250 (a CCITT subset of Estelle) in Veda", Protocol Specification, Testing, and Verification, V, editado por M. Diaz, North Holland, pp. 315-331.

FERNEDA, E., "Um compilador para a técnica de descrição formal Estelle/83", mestrado, DSC/CCT/UFPb, defendida em 09/mai/88.

GERBER, G. W. (1983), "Une méthode d'implantation automatisée de systèmes specifies formellement", tese de mestrado, IRO/UdeM, Montreal (Canadá).

ISO IS 7185 (1983), "Programming Language Pascal".

ISO TC97/SC16/WG1 subgroup B (1983), "A FDT Based on an Extended State Transition Model", Working Document.

ISO IS 7498 (1984), "Information Processing Systems - Open Systems Interconnection - Basic Reference Model".

ISO IS 9074 (1988), "Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model".

SOUZA, J.N. (1990), "Uma metodologia para validação, através de simulação, de especificações formais de protocolos de comunicação", mestrado, DSC/CCT/UFPb, defendida em 26/mar/90.

VUONG, S.T.; LAU, A.C. and CHAN, R.I. (1986), "Semiautomatic implementation of protocols using an Estelle-C compiler", IEEE Transactions on Software Engineering, Vol 14, Nº 3, março de 1986, pp. 384-393.

Anexo 1

Trecho da especificação Estelle/83 do protocolo Abracadabra

```
01  module ABRA;
02  end ABRAA;
03
04  refinement ABRA_BODY for ABRA;
05
06  const
07  ...,
08  type
09        channel MSAP (user, provider);
10              by user
11                    UnitReq (UnitData : UnitDTyp);
12              by provider
13                    UnitInd (UnitData : UnitDTyp);
14        end MSAP;
15
16        channel USAP (user, provider);
17              by user
18                    ConReq, ConResp, DatReq (UserData:UserDTyp);
19                    DisReq;
20              by provider
21                    ConInd, ConConf, DatInd (UserData: UserDTyp);
22                    DisInd;
23        end USAP;
24
25        channel PEERCODE (all);
26              by all
27                    CR, CC, DT (seq : seqtype ; UserData : UserDtyp),
28                    AK (seq:seqtype) , DR , DC;
29        end PEERCODE;
30
31        module STATION;
32              IpUser : USAP (provider);
33                    peer : PEERCODE (all);
34        end STATION;
35
36        process STATION_PROC for STATION;
37        ...;
38        trans
39              when IpUser . ConReq
40              from CLOSED
41              to CRSENT
42                    begin
43                          INITVAR;
44                                out peer . CR;
45                    SETTIMER (P);
46                    CRRetranRemaining : = - 1
47              end;
48        ...;
49        provided (CRRetranRemaining > 0) and (TIMEOUT)
50        from CRSENT
51        to CRSENT
52                    begin
53                      CRRetranRemaining : = CRRetranRemaining-1;
54                      out peer, CR
```

```
55                      end,
56                      ...;
57          end  STATION_PROC;
58
59          module  TRANSCODE;
60              Up  :  PEERCODE(all);
61              down  :  MSAP(user);
62          end  TRANSCODE;
63
64          process TRANSCODE_PROC  for  TRANSCODE;
65          ...;
66          end  TRANSCODE_PROC;
67
68          XC  :  TRANSCODE  with  TRANSCODE_PROC;
69          S   :  STATION  with  STATION_PROC;
70
71          connect  S.peer  to  XC.IpUp;
72      end  ABRA_BODY;
```

Anexo 2
Código Pascal relativo à especificação do Anexo 1

```
01  segment PROSAI;
02
03  const
04    ...,
05
06  type
07    ...;
08    channel = (C1MSAP, C2USAP, C3PEERCODE, C0STATION_PROC);
09    S1MSAP = (S1UnitReq, S1UnitInd);
10    S2USAP = (S2ConReq, S2ConResp, S2DatReq, S2DisReq,
11              S2ConInd, S2ConConf, S2DatInd, S2DisInd);
12    S3PEERCODE = (S3CR, S3CC, S3DT, S3AK, S3DR, S3DC);
13    S0STATION_PROC = (R1ANY);
14    ...,
15
16  var
17    P0VAR : P1TYPE;
18    C0VAR : C1TYPE;
19    S0VAR : S1TYPE;
20
21  procedure STATION_PROC;
22
23  label 1,
24
25  var
26    C1VAR : C1TYPE:
27    S1VAR : S1TYPE:
28
29  procedure INITVAR;
30  begin
31  ...;
32  end;
33
34  begin
35    C1VAR : = C0VAR;
36    S1VAR : = S0VAR;
37    with P0VAR —> . D0STATION_PROC do
38      begin
39        if C0VAR —> . IDENT = 1 then
40          if S0VAR —> . T2USAP = S2ConReq then
41            if STATE in ( . CLOSED . ) then
42              begin
43                STATE : = CRSENT;
44                INITVAR;
45                begin
46                  P0PROCEDURE(2);
47                  new (S0VAR, C3PEERCODE, S3CR);
48                  S0VAR —> . T3PEERCODE : = S3CR;
49                  SETTIMER (P);
50                  out
51              end;
52                CRRetranRemaining : = N-1;
53                goto 1
54              end;
```

```
55
56      ...;
57      if  C0VAR —> . IDENT  =  0  then
58        if  S0VAR —> . T0STATION_PROC  =  R2ANY  then
59          if  (CRRetranRemaining > 0)  and (TIMEOUT)  then
60            if  STATE  in ( . CRSENT . )  then
61              begin
62                STATE  :  =  CRSENT;
63                CRRetranRemaining  :  =  CRRetranRemaining-1;
64                begin
65                  P0PROCEDURE (2);
66                  new (S0VAR, C3PEERCODE,  S3CR);
67                  S0VAR —> . T3PEERCODE  :  =  S3CR;
68                  out
69                end;
70                goto  1
71              end;
72
73      ...;
74
75  end;
76
77  procedure  TRANSCODE_PROC;
78  ...;
79  begin
80  ...;
81  end;
82
83  procedure  I0ABRA_BODY (var  P1VAR  :  P1TYPE);
84
85  var
86    P2VAR  :  P1TYPE;
87    A0VAR,  A1VAR  :  A0TYPE;
88
89  procedure  I0STATION_PROC (var  P1VAR  :  P1TYPE);
90
91  begin
92    new (P0VAR,  P0STATION_PROC);
93    P1PROCEDURE  (STATION_PR;  P1VAR);
94    P3PROCEDURE  (FALSE, 0);
95    P3PROCEDURE  (TRUE, 1);
96    P3PROCEDURE  (TRUE,2);
97    with  P0VAR —> . D0STATION_PROC  do
98      STATE  :  =  CLOSED
99  end;
100
101 procedure  I0TRANSCODE_PROC (var P1VAR  :  P1TYPE);
102
103 begin
104   new (P0VAR,   P0TRANSCODE_PROC);
105   P1PROCEDURE (TRANSCOD_P;   P1VAR);
106   P3PROCEDURE (TRUE, 1);
107   P3PROCEDURE (TRUE, 2);
108 end;
109
110 begin
111   P2VAR  :  =  nil;
```

```
112     I0TRANSCODE_PROC (P2VAR);
113     I0STATION_PROCC (P2VAR);
114     P7PROCEDURE (P2VAR,  P1VAR);
115     P5PROCEDURE (2,  2,  P1VAR):
116     P6PROCEDURE (A0VAR);
117     P5PROCEDURE (1 ,  1 ,  P1VAR);
118     P6PROCEDURE (A1VAR);
119     A0VAR.CHAN —> . TARGET  : =  A1VAR;
120     A1VAR.CHAN —> . TARGET  : =  A0VAR;
121     P8PROCEDURE (P1VAR)
122   end;
```