CSP* : UM DIALETO DE CSP E SUA APLICAÇÃO À ESPECIFICAÇÃO DE PROTOCOLOS DE COMUNICAÇÃO (%)

*J.B.M.Sobral** J.L.S.Leão** A.C.P.Pedroza**

*CEC/UFSC
Caixa Postal 476
88049 Florianópolis
SC

**COPPE/UFRJ
Programa de Engenharia Elétrica
Caixa Postal 68504
21945 Rio de Janeiro, RJ

RESUMO

Este artigo mostra uma linguagem de especificação formal denominada CSP*, um dialeto de "COMMUNICATING SEQUENCIAL PROCESSES", que pode ser aplicada à especificação formal de protocolos de comunicação. Como exemplo, o protocolo ABRACADABRA da ISO/CCITT é especificado considerando-se o conceito de comportamento de processos característico desta linguagem, não se utilizando explicitamente o conceito tradicional de estado.

ABSTRACT

This paper presents the CSP* language, a dialect of CSP - Communicating Sequential Processes - applied to the formal specification of communication protocols. The ISO/CCITT ABRACADABRA protocol is specified using behavior oriented modelling, a characteristic of this language; thus the traditional state concept is not explicitly used.

(%) Trabalho realizado com auxílio da CAPES/PICD
* CEC/UFSC - Dept. de Ciências Estatísticas e da Computação
Universidade Federal de Santa Catarina

---

1 - INTRODUÇÃO

Existem muitas abordagens para especificação formal, empregando diferentes notações e estilos. Por conveniência, consideramos dois grupos, embora exista alguma sobreposição entre esses, já que alguns métodos se utilizam de técnicas de ambos os grupos. O primeiro grupo é o dos **métodos axiomáticos** e o segundo, o grupo dos **métodos baseados em modelos.** Dentre este último estão os métodos de especificação baseados em **modelos de transição** e os métodos baseados em **modelos algébricos.** Em ambos os casos podemos ter especificações na forma gráfica (estados e transições) ou especificações na forma textual de linguagem (descrições de comportamento), que seguem o estilo **imperativo** ou **declarativo** para linguagens de especificação.

Duas principais abordagens de especificação têm sido usadas para especificação formal de protocolos de comunicação: os modelos de transição de estados e os modelos orientados à comportamento que são baseados nas comunicações observáveis e nas abstrações dos eventos internos numa especificação.

CSP é uma linguagem de especificação formal voltada para a concorrência e comunicação, que teve considerável influência na definição de várias linguagens para sistemas concorrentes (ADA, OCCAM, e LOTOS, por exemplo). Uma escola de pensamento bem desenvolvida sobre concorrência e comunicação está centrada sobre a linguagem CSP. A versão original desta linguagem aparece em [HOARE 78]: em trabalhos subsequentes a natureza da linguagem mudou radicalmente. A comunicação binária foi acrescentada sincronização múltipla [CHARLESWORTH 87]. Comunicações de valores podem ser consideradas formalmente como um conjunto predefinido de ações (eventos), e uma abordagem algébrica para a teoria de sistemas de comunicação, é seguida em CSP. Portanto, a importância em estudar suas características e propriedades.

Este trabalho apresenta um formalismo de especificação denominado CSP*, que é baseado no modelo algébrico de CSP [HOARE 85]. Os formalismos de especificação baseadas em modelos algébricos permitem modelagens de sistemas concorrentes orientadas através do conceito de **comportamento**, não considerando explícitamente o conceito de **estado**. A nova versão desta linguagem, agora chamada TCSP (Theoretical CSP), consiste de um conjunto de operadores, junto com definições recursivas de processos. Esta linguagem e sua semântica é apresentada em [BROOKES 84], e [HOARE 85] dá a base completa da linguagem.

Algumas alterações foram feitas em CSP, que originaram o dialeto CSP* para aplicação a protocolos de comunicação. Consideramos, principalmente, as construções de CSP* para a descrição de comportamentos de objetos, numa linguagem textual de estilo **imperativo**, voltada à especificação.

O restante deste artigo está estruturado como segue: a seção 2 destaca as características da linguagem; a seção 3 mostra as alterações que originaram o dialeto; a seção 4 mostra sua aplicação ao protocolo ABRACADABRA; a seção 5 sugere um método de verificação automática; e finalmente na seção 6, estão alguns comentários sobre a linguagem e uma breve comparação com as linguagens ESTELLE e LOTOS.

2 - A LINGUAGEM CSP*

2.1 - O modelo de processos

A especificação de um protocolo em CSP* é construída de processos. Um processo em CSP/CSP* é uma descrição do comportamento de um objeto. Em CSP, um objeto pode ter características físicas. Para a nossa aplicação, os objetos são processos de um sistema de computação. Cada processo é especificado em duas partes: uma parte declarativa relativa à descrição de seus aspectos estáticos, e outra parte relativa à descrição de seu comportamento dinâmico, expressa pelo efeito que cada evento tem sobre o comportamento do objeto. Os processos para se comunicarem, explicitamente, devem ser conectados por um canal de comunicação, definido para CSP* como **bidirecional**. Um processo pode possuir subprocessos que relacionados por operadores constituem as construções básicas da linguagem. Basicamente, o comportamento de um objeto é descrito através de suas interações com o seu ambiente, isto é, os eventos que ocorrem ordenadamente nos canais compartilhados pelos processos e que constituem o comportamento observável. Embora o modelo seja baseado no princípio da observabilidade, algumas vezes, a descrição ou uma representação de eventos internos a um processo é necessária, porque eles podem influenciar o comportamento observável do processo [BRINKSMA 85].

2.2 - Aspectos Sintáticos e Semânticos

Esta seção contém um sumário da notação dos objetos sintáticos de CSP*, mostra a semântica informal e exemplifica as construções mais evidentes para utilização em protocolos. As formas sintáticas estão descritas numa notação BNF modificada, quando for conveniente para este texto.

2.2.1 - Símbolos Lógicos e Extralógicos

```
Veracidade/Falsidade          :  true , false
Conectivos lógicos            :  ^   v   ¬
Pontuação                     :  ,   .   (   )   :
Funcionais                    :  +   -   /   x   mod
Predicativos                  :  =   <>   <   >   <=   >=
Eventos                       : palavras com letras minúsculas
Processos                     : palavras com letras maiúsculas
Variáveis denotando eventos   :  x   y   z   ...
Variáveis denotando processos :  X   Y   Z   ...
```

2.2.2 - Estruturação de uma especificação

O formato de uma especificação CSP* é dada pelas descrições de todos os processos que compõem a especificação

global de um protocolo. Uma descrição de processo está formalizada como:

```
descrição  ::=  <declaração>  <definição>

declaração     ::=  nome_processo declarations
                                <texto formal declarativo>
                    end declarations

definição ::=  nome_processo == ( <definição_comportamento> )
==  significa  é definido como

definição_comportamento ::= (<inicialização> ; <construção>)

inicialização  ::=  ( {0 ; <atribuição>} ; SKIP )

atribuição  ::=  variável := <expressão de valor>

expressão de valor  ::= contantes ¦ variáveis ¦ funções

construção  ::=          <prefixação>
                      ¦ <composição_sequencial>
                    ¦ <condicional>
                  ¦ <iteraça~o determinisstica>
                ¦ <composição paralela>
              ¦ <intercalação>
            ¦ <escolha de processo>
          ¦ <iteração não-determinística>
        ¦ <restrição de eventos internos>
      ¦ <desabilitação de um processo>
```

-Texto formal declarativo

Os aspectos estáticos envolvidos nos processos, tais como os tipos de dados, não são definidos formalmente nesta linguagem. Para que seja possível uma completa descrição desses processos, é necessário complementar esta linguagem de especificação. CSP pressupõe determinados tipos de dados e acrescenta um texto formal declarativo para os aspectos estáticos dos processos, consistente com o nível de abstração da linguagem e com seu estilo imperativo de especificação. Uma notação próxima à linguagem ESTELLE [ESTELLE 87], é adotada para descrições de declaração.

Alguns tipos primitivos são considerados em CSP* : bool, nat, int, real, char, string, channel e timer.

Em CSP*, um canal é definido como uma variável do tipo primitivo "channel". Um outro tipo primitivo "timer" pode ser declarado para definir um processo ou conjunto de processos temporizadores. Uma declaração de canal em CSP* é semelhante à ESTELLE ressaltando os papéis de usuário e provedor, para os processos envolvidos na comunicação. As interações válidas são descritas com seus respectivos parâmetros. O exemplo seguinte mostra a definição de um canal, através de uma variável c.

```
var  c : array[0..1] of channel (Papel1,Papel2)
            by Papel1        : i1(par_1:x-type) ;
            by Papel2        : i2(par_a:y-type,par_b:z-type) ;
            by Papel1,Papel2: it1,it2 ;
```

A declaração acima indica que determinados processos P1 e P2 se comunicam pelo canal bidirecional c, onde P1 funciona como PAPEL1 e P2 como PAPEL2. No terceiro papel, CSP* provê a declaração de canal para o caso de processos que possuem as mesmas interações. O escopo para os parâmetros especificados nas interaçõoes só se dá no ato da comunicação.

CSP* tem uma forma sintática para declarar quais processos estão descritos numa especificação, com seu respectivo canal de comunicação e o seu papel na utilização desse canal. O exemplo abaixo mostra isto:

Process

```
USU_0.c_ucep[0]    : User ;
USU_1.c_ucep[1]    : User ;
ABRAC_0.c_ucep[0] : Provider ;
ABRAC_0.c_mcep[0] : User ;
ABRAC_1.c_ucep[1] : Provider;
ABRAC_1.c_mcep[1] : User ;
MEDIUM.c_mcep[0] & c_mcep[1] : Provider ;
DATA_TRANSFER.c_clock : User;
CLOCK.c_clock : Provider;
```

A notação "." significa "deposita interações no canal" e as palavras "user" e "provider" representam os papéis dos processos sobre o canal.

Uma definição de comportamento como:

USU_0 || ABRAC_0 || MEDIUM || ABRAC_1 || USU_1 indica quais processos estão conectados por "rendez-vous".

Elementos do tipo primitivo "timer" são usados para temporizar eventos e constituem processos que são declarados como um "array" em PASCAL. As funções são declaradas num texto formal declarativo com a seguinte sintaxe:

```
Abstração de função:
function nome({1,<formal>:<tipo_parâmetro>}):<tipo_função>
formal                ::= nome_parâmetro
tipo_parâmetro  ::= bool | int | real | string
tipo_função  ::= bool | int | real | string
Chamada de função  ::=  nome_função ( {1 , <atual>} )
atual                 ::= nome_parâmetro
```

- Descrição do Comportamento de Objetos: Processos

Para descrever modelos de comportamento, decide-se primeiro quais os eventos ou ações serão de interesse. O conceito geral de **evento** é o de uma ação atômica sem duração, cuja ocorrência pode requerer participação simultânea de mais de um processo. O conjunto de nomes de eventos que são considerados relevantes para a descrição particular de um **objeto** é chamado seu **alfabeto**. É logicamente impossível para um objeto engajar-se num evento fora desse conjunto. Um processo pode ser descrito em termos do conjunto finito de eventos do seu alfabeto. Para a nossa aplicação, o alfabeto de um processo corresponde as interações deste com o seu ambiente.

Algumas definições de processos especiais são importantes para descrição do comportamento de processos em CSP*. Por exemplo, STOP é o processo definido com um determinado alfabeto, que nunca se engaja com qualquer dos eventos constantes nele; ele representa, por exemplo, um objeto quebrado ou um "deadlock". SKIP é definido como um processo que nunca se engaja com os eventos de um alfabeto, mas termina sempre bem sucedido; ele representa a terminação bem sucedida de um processo sequencial, que pode ter uma continuação.

- Sintaxe para processos

```
processo  ::=  STOP | SKIP | <construção> | <instância>
instância  ::=  nome_processo ( {1 , variável} )
```

- Inicialização de Processos

Os processos em CSP* podem ter comportamentos definidos contendo um subprocesso para inicialização de objetos matemáticos. Um tal subprocesso pode conter:

(a) definição de constantes;
(b) inicialização de variáveis;
(c) criação de instâncias de processos.

A instanciação de processos correspondem aos processos a serem executados e estabelecem os canais de comunicação que são utilizados.

- Escopo e visibilidade de variáveis

CSP* tem regras de visibilidade e escopo, definidos de forma muito simples. Variáveis existentes dentro da parte declarativa são locais ao processo. O escopo associado à variável se estende através da definição do processo (descrição do comportamento), e a variável é visível dentro desse escopo. Uma variável não é visível dentro da definição de um processo subordinado chamado pelo processo. Um tal processo subordinado deve ter a sua própria parte declarativa, onde suas variáveis são declaradas. Em CSP* não existem variáveis compartilhadas. A única excessão são os canais que representam a única interface visível entre os processos.

Numa especificação de protocolo, a primeira parte declarativa e a primeira definição de comportamento devem conter a especificação global da arquitetura de comunicação. Neste caso, o escopo das declarações é a definição do conjunto de processos paralelos que fazem parte do sistema completo de comunicação.

2.2.3- Comunicação : entrada ou saída

As entidades de protocolos em camadas distintas trocam interações que são representadas por primitivas de comunicação. Uma entidade pode enviar uma interação à outra entidade adjacente, desde que exista um canal de comunicação declarado para tal. O conceito de canal em CSP é unidirecional, ao passo que em CSP* é bidirecional.

Os comandos de entrada e saída ? e ! são usados para especificar o recebimento e o envio de mensagens, respectivamente. O evento c ? i indica que um processo com o canal c recebe o valor de uma interação i de outro processo. O evento c ! i indica que um processo com o canal c envia o valor de uma interação i para outro processo, considerando passagem de parâmetros. O evento c.i corresponde à comunicação entre dois processos quando esses apenas se sincronizam e não há passagem de parâmetros. Sintáticamente:

```
entrada  ::= canal ? <interação> | canal.<interação>
saída    ::= canal ! <interação> | canal.<interação>
interação ::= nome_interação ( {@,parâmetros} )
```

Um "rendez-vous" em CSP é o mecanismo básico de sincronização e comunicação entre os processos. CSP processa sincronização e comunicação através dos comandos de entrada (?) e saída (!) que são usadas para estabelecer um "rendez-vous". O mecanismo de "rendez-vous" de CSP* é baseado no de CSP, mas tem a característica do "rendez-vous" de ESTELLE* [COURTIAT 88], onde as interações contendo parâmetros fortemente tipados são consideradas. O conceito de "rendez-vous" em CSP* é de uso amplo, tanto para comunicação externa à uma entidade, como para comunicação interna à mesma.

Uma guarda G é uma fórmula lógica que seleciona aquele comportamento no qual seu comando associado C, se aplica. Um comando guardado é executado e somente se, a avaliação de sua guarda é verdadeira. Um comando guardado avalia primeiro, sua guarda G e então o comportamento C é executado. Se a avaliação é falsa, o comportamento de G -> C é como SKIP.

Para expressarmos uma sincronização explicitamente entre dois processos, um pedido de sincronização aparece somente ao nível de comandos guardados e deve existir no máximo um pedido de sincronização por guarda.

Uma comunicação entre dois processos P1 e P2 é realizada pela passagem dos valores dos parâmetros de uma interação, dentro de uma instância do processo P1 à uma

realizada pela passagem dos valores dos parâmetros de uma interação, dentro de uma instância do processo P1 à uma intância do processo P2. Nenhuma outra comunicação poderá ser realizada enquanto as instâncias dos processos não terminarem o "rendez-vous".

Em CSP*, a notação c.i é usada quando os processos precisam ser sincronizados e não há comunicação de valores através de parâmetros. Neste caso, a interação é declarada a ambos os processos, e esses utilizam esta mesma notação quando de um pedido de sincronização.

Sejam dois processos P1 e P2, com pedidos de sincronização, respectivamente, c!i e c?i, que se comunicam através de um canal c. Estes dois processos podem ser sincronizados, se e somente se, as seguintes condições ocorrem:

- Um canal c é definido para P1 e P2 se comunicarem diretamente;
- A interação especificada em c!i é idêntica à especificada em c?i.
- Os comportamentos de P1 e P2 especificam simultaneamente, e respectivamente, as operações de saída (c!i) e entrada (c?i).

2.2.4 - As construções de CSP*

As seguintes construções são importantes para a especificação de um protocolo. Aqui são mostradas as formas sintáticas; a semântica informal completa para cada construção está em [HOARE 85]. No que segue, P e Q são processos, x é uma variável denotando um evento, b é uma expressão lógica, e é uma expressão de valor, G é uma guarda, C é uma definição de comportamento, e E é um subconjunto finito de eventos do alfabeto de um processo.

- Prefixação (x -> P) : Indica "x então P".

```
guarda ::= <evento> | boolean & entrada | boolean
evento ::= entrada | saída
prefixação ::= ( guarda -> nome_processo | (processo))
```

Exemplo:

```
( - - estabelecendo conexão : iniciador
  ( c_ucep[I] ? conreq ->
    ( E := 0 ; R := 0 ;
      ( c_mcep[I] ! unitreq(BuildCR) ->
        ( c_clock ! starttimer(P) -> WFCC)
      ) ) )
```

- Composição Sequencial (P ; Q) : É um processo que primeiro se comporta como P; quando P termina bem sucedido, (P ; Q) continua e comporta-se como Q. Genèricamente,

```
composição_sequencial ::= ({2 ; nome_processo} | (processo) )
```

Como exemplo, (x := e ; P) é um processo que se comporta como P, exceto que o valor inicial de x é definido ser o valor inicial da expressão e. Uma atribuição não é uma guarda e sim um processo que é componente de uma composição sequencial. Um outro exemplo é como segue:

```
ABRAC == ( CONNECTION_OPENING ;
           DATA_TRANSFER ;
           CONNECTION_CLOSING )
```

- Condicional (P if b else Q) : Indica uma escolha entre processos a ser realizada em função do valor lógico assumido pela expressão b.

Exemplo:

```
( c_clock ? a_timeout ->
  - - retransmite
  c_mcep[I] ! unitreq(Emission_Buffer) ->
  (Retrans_Attempts := Retrans_Attempts + 1;
   WFAK ) ) )
if  Retrans_Attempts < N  else  ERROR  )
```

que pode ocorrer sincronização e comunicação direta entre eles; requer a interação entre P e Q, definindo, implicitamente, o "rendez-vous" de CSP*, quando um evento pertence a ambos os alfabetos. É previsto em CSP e CSP*, paralelismo síncrono entre processos assíncronos. Em CSP e CSP*, não existem interações assíncronas. Um "rendez-vous" está implícito nesta notação. Genèricamente:

```
processos_paralelos ::= { 2 || nome_processo : (processo) }
```

Exemplo: (USU_0 || ABRAC_0 || MEDIUM || ABRAC_1 || USU_1)

- **Escolha de processo (P [] Q) :** Indica uma escolha determinística ou não-determinística de processo, quando o ambiente pode controlar qual de P ou Q será escolhido. Uma estrutura baseada no operador [] pode ser construída através de comandos guardados, cada um compreemdendo uma guarda e uma descrição de comportamento associado. A avaliação das guardas proporcionará a seleção de um comportamento a ser executado. A semântica informal para esta construção está em [MORGAN 90].

```
Exemplo:
- - esperando por resposta do usuário  (conresp)
WFUR == ( ( c_ucep[I] ? conresp ->
            ( E := 0 ; R := 0 ;
              c_mcep[I] ! unitreq(BuildCC) -> DATA_TRANSFER))
        []
          ( c_ucep[I] ? disreq ->
            ( c_mcep[I] ! unitreq(BuildDR) ->
              CONNECTION_CLOSING ) )
        []
          ( c_mcep[I] ? unitind ->
            ( c_ucep[I] ! disind ->
              c_mcep[I] ! unitreq(BuildDC) ->
              CONNECTION_OPENING )
            if Code(pdu) = DR else  SKIP ) )
```

- **Intercalação (P ||| Q) :** Combina processos com o mesmo alfabeto, para operarem concorrentemente, sem interagirem ou se sincronizarem diretamente. Ao contrário de ||, este operador introduz escolha não-determinística entre os processos, se ambos podem engajar-se num mesmo evento.

```
Exemplo 1:  MEDIUM == ( PIPE_01 ||| PIPE_10 )

Exemplo 2:
R_ABRAC == ( ((Code(pdu) = DT) ^ (Seq(pdu) = R)) ->
               ( R := (R + 1) mod 2 ;
                 ((( c_ucep[I] ! dataind(sdu) -> SKIP )
                     |||
                   ( c_mcep[I] ! unitreq(BuildAK(R)) ->
                     SKIP )) ;
                  (DATA_TRANSFER [] WFAK)  - - depende
                  de onde estava quando entrou em
                  R_ABRAC ))
           ... ... ...
```

- **Iteração determinística :**

"loop" : *P == P;P;P;...

"while" : (G) *(processo), onde (processo) representa o comportamento de exatamente um processo;

```
Exemplo:  (t < T) *TIMING )
```

definição recursiva de processo: É o processo X com alfabeto A tal que X == F(X); definindo-se recursão por equação, temos por exemplo, C == (x -> (y -> C)) , onde o alfabeto de C é {x,y}.

```
Exemplo:
BUFFER == ( ( c_mcep[I] ? unitreq(pdu) ->
              ( ( c_mcep[J] ! unitind(pdu) -> BUFFER )
              [] BUFFER - - quando o meio perde a pdu ) )
```

- Iteração Não-Determinística : Como no caso de uma estrutura escolha, também é construída de comandos guardados e a repetição ocorre enquanto houver uma ou mais guardas abertas, existindo neste último caso, uma escolha não-determinística entre guardas, selecionando-se assim um corportamento C associado.

```
Exemplo:   do  G1 -> C1
           [] G2 -> C2
           [] G3 -> C3
           od
```

- Desabilitação de processo (P^ Q) : Em quase todos os protocolos ou serviços orientados à conexão, pode ocorrer que o curso normal de ação de uma entidade de protocolo seja interrompido em qualquer instante do tempo, por eventos sinalizando desconexão ou aborto de uma conexão. CSP* define um tipo de composição sequencial que não depende de terminação bem sucedida de P. O progresso de P é interrompido na ocorrência do primeiro evento de Q, e P nunca será retomado.

- Restrição (P \ E) : Oculta eventos no alfabeto de P, que não podem ser observados ou controlados pelo ambiente de P. Em geral, o alfabeto de um processo contém exatamente aqueles eventos que são considerados importantes. e cuja ocorrencia requer a participação simultânea do ambiente do processo. Na descrição do comportamento interno de um processo, frequentemente necessitamos considerar eventos representando transições internas do processo. Após a descrição do comportamento do processo, pode-se ocultar todas as ocorrências de eventos internos ao processo. O que se pretende é que estes eventos ocorram automaticamente e instantâneamente sem serem controlados pelo ambiente do processo.

2.2.5 - Paralelismo e Tempo

Para temporizar eventos, CSP* utiliza-se da declaração, acesso e resposta de processos temporizadores do tipo "timer". Um processo temporizador numa especificação é executado através de uma instanciação da seguinte descrição de comportamento, de um processo identificado por CLOCK:

```
CLOCK (K) == ( c_clock[K] ? starttimer(T) ->
                ( ( t := 0 ; (t < T) *TIMING );
                  ( c_clock[K] ! a_timeout ->
                    CLOCK (K) ) ) )

TIMING = ( ( c_clock[K] ? stoptimer -> CLOCK (K) )
           [] ( t := t + 1 ; SKIP ) )
```

CSP* considera que processos temporizadores definidos desta forma são acessados e proporcionam resposta, através de canais bidirecionais, aqui denominados c_clock[K], e são instanciados pela passagem dos parâmetros T, F e a referência ao canal pela variável K. As forma sintáticas para acesso e resposta de um processo temporizador são como segue:

```
entrada        ::= input_timer
input_timer    ::= canal[K] ! starttimer(Tempo_Max)
Tempo_Max      ::= valor_tempo_máximo_temporização
saída       ::= output_timer
output_timer   ::= canal ? a_timeout
parada         ::= stop_timer
stop_timer     ::= canal[K] ! stoptimer
K              ::= índice de canal
```

Estas formas são especificadas na definição de comportamento de um processo requerendo um serviço de temporização. Formas duais para entrada, saída e parada são especificadas no processo temporizador, como mostra o processo CLOCK. Um "input_timer" significa que um temporizador recebe uma interação representando um tempo máximo durante o qual um evento deve ocorrer. Processos acessando temporizadores podem receber "timeout" como segue: um "output_timer"

interação representando um tempo máximo durante o qual um evento deve ocorrer. Processos acessando temporizadores podem receber "timeout" como segue: um "output_timer" representa que um temporizador pode emitir uma interação "a_timeout" significando um "timeout" assíncrono para um determinado evento, representando o esgotamento de um tempo máximo estabelecido. Processos dependentes de tempo podem parar temporizadores através de uma interação "stop_timer" significando que o evento esperado por um processo ocorreu antes de um "timeout", e o processo sinaliza a parada do temporizador.

3 - AS ALTERAÇÕES EM CSP

As alterações sintáticas e semânticas previstas em CSP* estão resumidas na tabela abaixo:

CSP		CSP*
não declara tipos	..	texto formal declarativo
=	..	==
canal ? variável	..	canal ? interação
canal ! valor	..	canal ! interação
-	..	canal . interação
canal unidirecional	..	canal bidirecional (declarado como variável)
-	..	construção para iteração não-determinística
"rendez-vous" (simples)	..	"rendez-vous" (simples) (c/ parâmetros tipados)
-	..	temporiza eventos

4 - EXEMPLO DE ESPECIFICAÇÃO: O PROTOCOLO ABRACADABRA

Nesta seção é apresentada uma parte da especificação do protocolo ABRACADABRA. Este protocolo é definido de maneira conjunta pelos grupos FDT da ISO e CCITT [ISO 86], e apresenta características importantes dos protocolos normatizados.

A figura 1 apresenta a arquitetura ABRACADABRA segundo um modelo em camadas, onde duas entidades ABRACADABRA interagem para oferecer um serviço às entidades usuárias, localizadas na camada imediatamente superior.

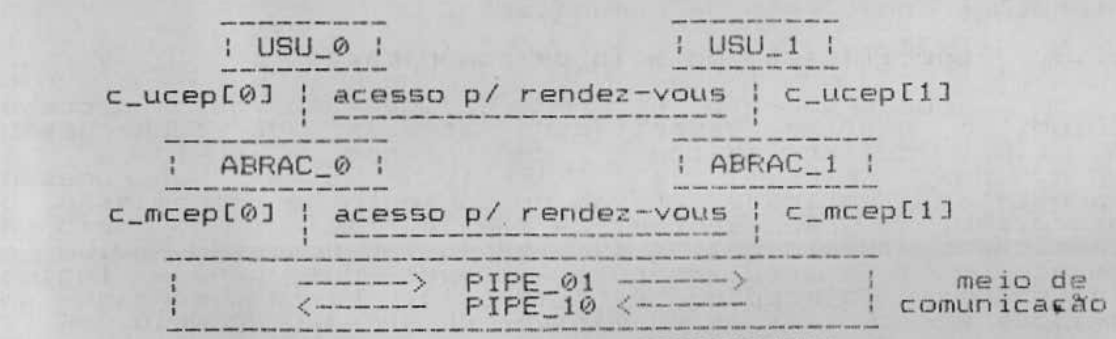

Figura 1 - Arquitetura ABRACADABRA

4.1 - Especificação Informal do Serviço e Protocolo

O serviço oferecido por uma entidade ABRACADABRA é orientado à conexão, bidirecional e simultâneo. Ele permite a dois usuários USU_0 e USU_1 trocar, uma vez que a conexão exista, unidades de dados de serviço (sdu's) de maneira confiável e com garantia de sequenciamento.

- Especificação do Serviço

As primitivas de serviço são respectivamente:

Fase de estabelecimento de conexão:

. conreq, conind, conresp e conconf;

. disreq e disind.

- Especificação do Protocolo
  Unidades de Dados do Protocolo (pdu's)
  . Fase de estabelecimento de conexão:
    CR e CC
  . Fase de transferência de dados:
    DT e AK
  . Liberação de conexão:
    DR e DC

- Constantes Definidas
  N - número máximo de tentativas de envio de DT-pdu sem recepção de reconhecimento;
  P - retardo antes de nova transmissão de DT-pdu no meio de comunicação;
  Max_Data - tamanho máximo de uma sdu ;

- Variáveis Definidas
  E - número de sequência da próxima DT a enviar;
  R - número de sequência da DT esperada;
  Retrans_Attempts - número de tentativas realizadas.

- Serviço de Comunicação
  . Não orientado à conexão;
  . Pode perder mensagens, ABRACADABRA temporiza e reenvia;
  . Não pode: desordenar, corromper, duplicar, e criar;
  . Primitivas de serviço: unitreq e unitind

4.2 - Especificação Formal em CSP*

4.2.1 - Especificação de uma entidade ABRACADABRA

Uma especificação do protocolo ABRACADABRA encontra-se no anexo. Nesta especificação, uma entidade ABRACADABRA, é especificada como um processo denominado ABRAC_ENTITY, e a especificação completa com todos as entidades e processos (usuárias, provedoras e meio) envolvidos, é identificada por CSP*_ABRAC.

4.2.2 - Especificando os canais do processo ABRAC_ENTITY

Um processo ABRAC_ENTITY se comunica com o seu ambiente através de dois canais bidirecionais denominados c_ucep[I] e c_mcep[I]. A variável I instancia os processos ABRAC_ENTITY, podendo assumir os valores 0 ou 1. Um canal c_ucep[I] envia e recebe primitivas de serviço de uma entidade ABRACADABRA e um canal c_mcep[I] envia e recebe as interações com o meio de comunicação.

4.2.3 - Especificação do meio de comunicação

O comportamento do meio é modelado pelo processo MEDIUM, o qual é especificado através de subprocessos do tipo "buffer-limitado" de tamanho igual a 1, denominados PIPE_01 e PIPE_10. Estes subprocessos representam entidades ativas no contexto de transmissão da informação, e representam os canais físicos do meio de comunicação bidirecional e simultâneo. O processo MEDIUM se comunica com o seu ambiente através dos canais lógicos bidirecionais c_mcep[0] e c_mcep[1]. Nestes canais são enviadas e recebidas as primitivas de serviço do meio.

4.2.4 - Observações sobre o exemplo

A especificação mostrada no anexo engloba as fases de estabelecimento, encerramento de conexão e a fase de tranferência de dados. Os nomes dos processos e subprocessos correspondem aos bem conhecidos nomes de estados, por uma questão de facilidade. Neste exemplo, procurou-se manter a conformidade com a terminologia da ISO, como está em [COUTIAT 87a] e [SAQUI 90]. As variáveis I e J especificam instâncias, quando a elas são atribuídas os valores 0 ou 1. Code(...) e Seq(...) são funções que avaliam o código de uma pdu e seu número de sequência, respectivamente. Build... é uma função que monta cada pdu a ser transmitida, de acordo com seu tipo de código. Para temporizar "timeouts",

pdu e seu número de sequência, respectivamente. Build... é uma função que monta cada pdu a ser transmitida, de acordo com seu tipo de código. Para temporizar "timeouts", é utilizada a descrição do processo CLOCK da seção 2.2.5 . A versão de CSP aqui utilizada é aquela dada em [HOARE 85], correspondente ao TCSP (Theoretical CSP).

5 - CSP* E A VERIFICAÇÃO DE PROTOCOLOS

Uma abordagem que utiliza um método axiomático de verificação automática de propriedades de programas concorrentes está em [FIALHO 89]. Esta abordagem de verificação expressa propriedades através de predicados da Lógica Temporal. O método tem como base para especificação formal e implementação dos sistemas concorrentes, a linguagem CRIS [LEÃO 89]. Esta linguagem é baseada no conceito de sistemas de transição, e como tal, a utilização do método fica facilitada para sistemas descritos nesta linguagem. O protótipo de uma ferramenta de verificação foi desenvolvido para este método.

Um traço do comportamento de um processo é uma sequência finita de símbolos registrando os eventos nos quais o processo se engajou até algum momento do tempo. Embora CSP* seja uma linguagem que segue o estilo orientado à comportamento, algumas vezes é útil fazer uma representação gráfica de um processo como uma estrutura de árvore consistindo de nodos que são associados à comportamentos (estados), e arcos correspondendo aos eventos do alfabeto do processo. O conceito de traço em CSP, torna viável a verificação de propriedades de segurança, de vivacidade e de precedência, para uma especificação CSP*, através deste método.

- Exemplo de Verificação:

O propósito do modelo OSI é definir o protocolo de uma camada (N) através do serviço da camada (N - 1). Um tal protocolo deve ser provado correto, através de métodos de verificação: uma prova formal de que, por interação via o serviço (N - 1), e de acordo com o protocolo (N), as entidades (N) realmente fornecem o serviço esperado.

Para exemplificar a aplicação do método, é utilizada a especificação do protocolo ABRACADABRA. A arquitetura do sistema que deve ser verificado é representada pelo processo CSP*_ABRAC e composta de dois processos que modelam os usuários (USU_0 e USU_1), dois processos que modelam as entidades ABRACADABRA (ABRAC_0 e ABRAC_1) e o processo que representa o meio de comunicação (MEDIUM), interconectados convenientemente em CSP* pela definição global do sistema, como segue:

```
(1)     CSP*_ABRAC == ( USU_0 || ABRAC_0 || MEDIUM ||
                        ABRAC_1 || USU_1 )
```

A especificação de uma entidade usuária USU_0 e USU_1 é:

```
USU_0 == ( I := 0 ; USER_ENTITY )
USU_1 == ( I := 1 ; USER_ENTITY )
USER_ENTITY = ( CLOSED [] WFCONCONF [] USERESP [] OPEN )

CLOSED == ( c_ucep[I]!conreq -> WFCONCONF   []
            c_ucep[I]?conind -> USERESP     []
            c_ucep[I]?disind -> CLOSED )

WFCONCONF == ( c_ucep[I]?conconf -> OPEN    []
               c_ucep[I]!disreq -> CLOSED   []
               c_ucep[I]?disind -> CLOSED )

USERESP == ( c_ucep[I]?disind -> CLOSED     []
             c_ucep[I]!disreq -> CLOSED     []
             c_ucep[I]!conresp -> OPEN )

OPEN == ( c_ucep[I]!datareq -> OPEN   []
```

```
ABRAC_0 == ( I := 0 ; ABRAC_ENTITY )
ABRAC_1 == ( I := 1 ; ABRAC_ENTITY )
MEDIUM  == ( PIPE_01 ||| PIPE_10 )
```

Como um primeiro exemplo, seja verificar um procedimento de abertura de conexão na qual um usuário envia um pedido de conexão, e um usuário remoto irá aceitar e confirmar o pedido, ou rejeitá-lo e encerrar o pedido de conexão.

O método de verificação apresentado em [FIALHO 89], descreve este procedimento como um predicado da Lógica Temporal, que é parte da entrada de dados para a ferramenta de verificação.

Para os nossos propósitos, convencionamos que a parte principal do comportamento global do sistema descrito em (1) é expressa pela notação:

[ABRAC_0, ABRAC_1, USU_0, USU_1] ,

e um predicado para o caso de rejeição do pedido de conexão pode ser especificado como:

```
at [WFCC,CONNECTION_OPENING,C2,C1] ->
◇ ( ~ (at[*,*,C4,*]) U at[*,*,C1,*] )
```

onde: C1 é o processo CLOSED
C2 é o processo WFCONCONF
C3 é o processo USERRESP
C4 e o processo OPEN

e ◇ operador temporal "eventualmente"
~ negação do predicado temporal
U operador temporal "até"

Este predicado indica que se um pedido de conexão foi enviado, a entidade ABRAC_0 se encontra no processo de comportamento WFCC, a entidade remota ainda está no comportamento CONNECTION_OPENING, o usuário requisitante USU_0 está no comportamento C2 e o usuário remoto ainda está no comportamento C1. Assim, partindo deste comportamento inicial C1, eventualmente (dentro de um intervalo finito) será alcançado um comportamento global onde o usuário requisitante volta ao comportamento inicial C1 (através do recebimento de um pedido de desconexão) sem ter passado pelo comportamento onde a conexão foi confirmada, ou seja [*,*,C4,*].

Um outro predicado, para o caso de aceitação do pedido de conexão pode ser expresso como:

```
at [WFCC,CONNECTION_OPENING,C2,C1] ->
◇ ( ~ (at[*,*,C1,*]) U at[*,*,C4,*] )
```

Um segundo exemplo, é verificar um predicado que garanta o envio de um dado, durante a fase de tranferência de dados. Este pode ser descrito como:

```
at [DATA_TRANSFER,DATA_TRANSFER,C4 and SDU=v,C4 and SDU=*]
-> ◇ at[DATA_TRANSFER,DATA_TRANSFER,C4 and SDU=*,
        C4 and SDU=v]
```

Neste predicado, uma SDU enviada por um usuário é eventualmente recebida pelo outro usuário, desde que não haja desconexão e o meio de comunicação seja capaz de transmitir corretamente ou perder a mensagem, já que o protocolo ABRACADABRA está preparado para reenviar uma mensagem perdida. Se o meio corrompe, duplica, cria ou altera a sequência em que mensagens são enviadas, então não se pode concluir pela garantia da transmissão.

Os seguintes dados devem ser fornecidos à ferramenta de verificação: a descrição em CRIS do protocolo equivalente à especificação CSP* (por enquanto é assim) a ser verificado,

o predicado que se deseja verificar, como também a heurística usada para determinar um caminho mais promissor a partir de alternativas em um determinado estado.

6 - COMENTARIOS FINAIS

Uma especificação de protocolos em CSP*, apresenta-se na forma textual de uma linguagem de especificação imperativa, onde as comunicações entre os processos formam a essência da linguagem. O estilo recomendado de escrita em CSP* baseia-se no conceito de comportamento, e não considera explicitamente o conceito de estado. A especificação de um protocolo em CSP* apresenta um nível de abstração aceitável. O protocolo pode ser especificado sob o ponto de vista de um observador externo. E possível descrever sua estrutura interna em termos de subprocessos, e como cada subprocesso se comporta sob o ponto de vista do seu ambiente. O acesso por "rendez-vous" garante a atomicidade dos eventos entre camadas, e possui um grau aceitável de abstração, visto de forma o mais transparente possível quando da especificação das comunicações.

A linguagem CSP* possuí alguns requisitos importantes de uma técnica de descrição formal para especificar modelos OSI. Entre esses podemos citar os seguintes: **expressividade**, pois qualquer sistema concorrente pode ser especificado, e propriedades relevantes à descrição de entidades OSI podem ser expressas nesta linguagem; **abstração** que é caracterizada pelo princípio da observabilidade e comunicação por "rendez-vous"; **não-determinismo**: a linguagem CSP* satisfaz à descrição de escolha entre diferentes comportamentos, numa forma externa, feita em cooperação com o ambiente do processo, através dos operadores [] e ||| ; e **modularidade**, que é suportada pela descrição dos elementos básicos numa especificação, em termos de processos, e a utilização dos canais como os únicos elememtos referenciados e compartilhados pelos processos numa comunicação. A característica modular de CSP*, não é tão completa quanto á modularidade apresentada por uma linguagem que se comunica através de portas, como por exemplo LOTOS, CCS e CRIS. Uma porta de comunicação é um objeto sintático que é declarado localmente a um processo. Uma porta, portanto, não é um objeto sintático compartilhado pelos processos, como é um canal que é declarado de forma global.

Agora esboçaremos uma breve comparação entre as técnicas de ESTELLE e LOTOS. Nos concentraremos sobre alguns aspectos de modelagem e conceitos semânticos, ao invés de comparações sintáticas. Comparativamente à ESTELLE, CSP* apresenta um nível mais alto de abstração. Paralelismo por intercalação pode ser expresso internamente a uma entidade (processo) de protocolo em CSP*, e não pode ser expresso em ESTELLE, que é sequencial neste caso. O único modo de comunicação em CSP/CSP* é por "rendez-vous", ao passo que em ESTELLE padrão é por filas FIFO. A comunicação por "rendez-vous" é melhor em termos de especificação e verificação, porque reduz o número de estados acessíveis que o protocolo pode assumir. O conceito de ponto de interação de ESTELLE inexiste em CSP*, que usa apenas o conceito de canal para comunicação. CSP* usa instanciação estática, enquanto ESTELLE se utiliza de primitivas para instanciação dinâmica como uma linguagem mais próxima da fase de implementação.

Comparativamente à LOTOS, todos os operadores da linguagem LOTOS básica [ISO LOTOS 87], surgiram de conceitos já existentes em CSP [HOARE 78], os quais estão aqui mostrados em CSP*. Portanto, o que distingue CSP* de LOTOS, quanto às definições de comportamentos de processos, é a sintaxe das linguagens. Semânticamente, os conceitos básicos são praticamente os mesmos. Veja em [BOLOGNESI 87]. Como em LOTOS, CSP* também faz uso de textos formais declarativos para os aspectos estáticos envolvidos nos processos. Entretanto, se utiliza de tipos de dados primitivos, como em ESTELLE, enquanto LOTOS, faz uso de tipos abstratos de dados, consistentemente com os requerimentos de abstração de detalhes de implementação apresentados pela linguagem.

linguagem. CSP* tem construções baseadas na arquitetura de máquina de Von Neumann, tais como variáveis, operação de atribuição e repetição; características estas de linguagens de programação imperativas. LOTOS evidencia mais o princípio da observabilidade e se abstrái de eventos internos de forma mais genérica, pois tem somente uma representação para todos os eventos internos. Portanto, ocultando as características imperativas de CSP*. Ambas as linguagens são baseadas em modelos algébricos que tem como elementos básicos os processos. A noção de estado é fundamental em ESTELLE, mas CSP* e LOTOS são baseadas em comportamentos, embora possam ser interpretadas como um sistema de transição rotulado.

A característica construtiva de CSP* pode conduzir a especificações adequadas como linguagens-fontes para compiladores e consequente execução da especificação.

A experiência com CSP* mostrou uma linguagem para descrição de sistemas concorrentes, baseada num modelo algébrico de processos, de estilo imperativo. Este trabalho é o primeiro de um estudo mais abrangente sobre técnicas de descrição formal para sistemas concorrentes. Diferentemente da abordagem imperativa da linguagem CSP*, é de interesse o estudo de especificação formal baseada num modelo matemático (lógico e/ou funcional), de estilo declarativo, voltado à rápida prototipagem e que suporte o desenvolvimento de uma ferramenta de verificação automática, cuja utilização fique facilitada para sistemas descritos nessa linguagem.

6 - REFERENCIAS BIBLIOGRAFICAS

[HOARE 78] Hoare, C. A. R., "Communicating Sequential Processes", Comm.ACM 21 (8), pp. 666-677 (1978).

[BROOKES 84] Brookes, S., Hoare, C., Roscoe, A.; "A Theory of Communicating Sequencial Processes", JACM, 31, N. 7, 560-599, 1984.

[HOARE 85] Hoare.C. A. R., "Communicating Sequential Processes", Prentice Hall, (1985).

[ISO 86] "Status and applicability of formal description techniques", ISO/TC 97/ SC 21 N1533, September (1986).

[BRINKSMA 85] Brinksma, E., Karjoth, G.; " A specification of the OSI Transport Service in LOTOS"; Protocol, Specification, Testing, and Verification IV, IFIP 1985.

[ISO LOTOS 87] "LOTOS, A formal Description Technique Based on the Temporal Ordering of Observational Behaviour", ISO DIS 8807, July 1987.

[BOLOGNESI 87] Bolognesi, T., Brinksma, Ed; "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems 14, 25-59, 1987.

[CHARLESWORTH 87] Charlesworth, A.; "The Multiway Rendezvous", ACM Transactions on Programming Languages and Systems, Vol. 9, N. 2, 350-366, 1987.

[ESTELLE 87] "ESTELLE, a Formal Description Thecnique based on an extend state transition model", ISO DIS 9074, June 1987.

[COURTIAT 87a] Courtiat, J. P., "Contribuition a la Description Formelle de Protocoles", These Docteur D'Etat, L'Université Paul Sabatier, Toulouse, (1987).

[COURTIAT 87b] Courtiat, J. P., "Proposition of a rendez-vous mechanism for ESTELLE", SEDOS report

[COURTIAT 88] Courtiat, J. P., "ESTELLE* : a powerful dialect of ESTELLE for OSI protocol description", 8th IFIP Symposium on Protocol Specificatiom, Testing and Verification, Atlantic City, June 1988.

[LEAO 89] Leão, J. L. S.; Fialho, S. V.; Pedroza, A.; "A State Based Language for Discrete Control and the Verification of its Programs", International Symposium on Circuits and Systems, Portland, Oregon, maio 1989.

[FIALHO 89] Fialho, S. V.; Pedroza, A.; Leão, J.; Junior, R. C. O.; "Uma Ferramenta Para Verificação Automática de Programas Concorrentes", Simpósio Brasileiro de Telecomunicações, Florianópolis,1989.

[MORGAN 90] Morgan, C.; Programming from Specifications Prentice Hall, 1990.

[SAQUI 90] Saqui-Sannes, P ; "Prototypage D'Un Environnement de Validation de Protocoles: Aplication a L'Approache Estelle" ; These L'Université Paul Sabatier, LAAS, Toulouse, 1990

ANEXO - ABRACADABRA COM RENDEZ-VOUS - CSP*

```
CSP*_ABRAC declarations
   const  Max_Data = 4 ;
          P = 10 ;
   type   data_type = ( ..... ) ;
          pdu_type  = ( ..... ) ;
   var
        c_ucep : array [0..1] of channel (User,Provider) ;
                    by User :
                       conreq;
                       conresp;
                       datareq ( sdu: data_type );
                       disreq;
                    by Provider :
                       conind;
                       conconf;
                       data_ind ( sdu: data_type );
                       disind;

        c_mcep : array [0..1] of channel (User,Provider) ;
                    by User : unitreq ( pdu: pdu_type ) ;
                    by Provider : unitind ( pdu: pdu_type ) ;

        c_clock : channel (User,Provider) ;
                    by User : starttimer (P:real) ;
                              stoptimer ;
                    by Provider : a_timeout ;

        Process

              USU_0.c_ucep[0]   : User ;
              USU_1.c_ucep[1]   : User ;
              ABRAC_0.c_ucep[0] : Provider ;
              ABRAC_0.c_mcep[0] : User ;
              ABRAC_1.c_ucep[1] : Provider;
              ABRAC_1.c_mcep[1] : User ;
              MEDIUM.c_mcep[0] & c_mcep[1] : Provider ;
              DATA_TRANSFER.c_clock : User;
              CLOCK.c_clock : Provider;

end declarations ;
CSP*_ABRAC ==
            (USU_0 || ABRAC_0 || MEDIUM || ABRAC_1 || USU_1)
.........
```

```
USU_0 == ( I := 0 ; USER_ENTITY )

.........

USU_1 == ( I := 1 ; USER_ENTITY )

.........

USER_ENTITY == ( CLOSED [] WFCONCONF [] USERESP [] OPEN )
          CLOSED    == ( .......... )
          WFCONCONF == ( .......... )
          USERESP   == ( .......... )
          OPEN      == ( .......... )

.........

ABRAC_0 == ( I := 0 ; ABRAC_ENTITY )

.........

ABRAC_1 == ( I := 1 ; ABRAC_ENTITY )

- - especificação do meio de comunicação
MEDIUM == ( PIPE_01 ||| PIPE_10 )

..........

PIPE_01 == ( I := 0 ; J := 1 ; BUFFER )

.........

PIPE_10 == ( I := 1 ; J := 0 ; BUFFER )

BUFFER == ( ( c_mcep[I] ? unitreq(pdu) ->
              ( ( c_mcep[J] ! unitind(pdu) -> BUFFER )
              [] BUFFER - - quando o meio perde a pdu ) )

- - Especificação de uma entidade ABRACADABRA
ABRAC_ENTITY declarations ;
    const N = 1 ;
    type  code_type = ( CR,CC,DR,DC,DT,AK ) ;
          pdu_type  = record
                        ...
                        ...
                        ...
                        ...

    var   Retrans_Attempts    : 0..N ;
          E                   : 0..1 ;
          R                   : 0..1 ;
          Emission_Buffer     : pdu_type ;
          Disconnection_Requested: bool ;

    - - Declarando um processo temporizador
    CLOCK : timer ;

    - - Declarações de funções
    function  BuildCR  : pdu_type;
    function  BuildCC  : pdu_type;
    function  BuildDR  : pdu_type;
    function  BuildDC  : pdu_type;
    function  BuildDT  (udata:data_type, nseq:int):pdu_type ;
    function  BuildAK  ( nseq  : int ) : pdu_type  ;
    function  Code     ( pdu   : pdu_type ) : code_type ;
    function  Seq      ( pdu   : pdu_type ) : int ;

end ABRAC_ENTITY declarations;

RESET_EMISSION_VARIABLES == ( Emission_Buffer.code := DT ; .
                              Emission_Buffer.n := 0 ;
                              Emision_Buffer.data ;= sdu ;
                            Disconnection_Requested := false ;
                            Retrans_Attempts := 0 )

ABRAC_ENTITY == ( INITIALIZE ; ABRAC )
INITIALIZE == ( E := 0 ; R := 0 ; RESET_EMISSION_VARIABLES )
```

```
ABRAC == ( CONNECTION_OPENING ; DATA_TRANSFER ;
                    CONNECTION_CLOSING  )

- -  Fase de estabelecimento de conexão
CONNECTION_OPENING == ( - - estabelecendo conexão : iniciador
                  ( c_ucep[I] ? conreq ->
                    ( E := 0 ; R := 0 ;
                      ( c_mcep[I] ! unitreq(BuildCR) ->
                        ( c_clock ! starttimer(P) -> WFCC)
                      ) ) )
                  []
                  ( - - estabelecendo conexão : responder
                    c_mcep[I] ? unitind ->
                    ((Code(pdu) = CR ) ->
                       ( c_ucep[I] ! conind -> WFUR )
                      []
                     (Code(pdu) = DR) -> CONNECTION_OPENING ))

WFCC   == ( c_mcep[I] ? unitind ->
            ( ((Code(pdu) = CC) v (Code(pdu) = CR)) ->
                ( c_ucep[I] ! conconf -> DATA_TRANSFER )   []
            ( (Code(pdu) = DR) ->
                (( c_ucep[I] ! disind -> SKIP)   |||
                 ( c_mcep[I] ! unitreq(BuildDC) -> SKIP ))
                ; CONNECTION_OPENING ) )   []
            ( c_ucep[I] ? disreq ->
              ( ( c_mcep[I] ! unitreq(BuildDR) ->
                  ( Disconnection_Requested := true ;
                    CONNNECTION_CLOSING ) ) )
            []
            ( ( c_clock ? a_timeout ->
                ( Retrans_Attempts := Retrans_Attempts + 1 ;
                  ( c_mcep[I] ! unitreq(BuildCR) ->
                    ( c_clock ! starttimer(P) -> WFCC ))
                 )
                 if  Retrans_Attempts < N  ^
                              ¬ Disconnection_Requested  else
                 (
                   Retrans_Attempts := Retrans_Attempts + 1
                   ->  WFCC )
                  if  Disconnection_Requested = true else
                  (  c_ucep[I] ! disind  ->
                     c_mcep[I] ! unitreq(BuildDR) ->
                     ( Disconnection_Request := true ;
                       RESET_EMISSION_VARIABLES ;
                       CONNECTION_CLOSING ) ) ) ) ) )

- - Fase de transferência de dados
DATA_TRANSFER ==
      (    - - envio de dados
           ( c_ucep[I] ? data_req(sdu) ->
             ( Emission_Buffer.data := BuildDT(sdu,E) ;
               ( c_mcep[I] ! unitreq(Emission_Buffer) ->
                 ( c_clock ! starttimer(P) -> WFAK )
               ) ) )
           []
           - - recepção de dados
           ( c_mcep[I] ? unitind -> R_ABRAC )
           []
           - - pedido de desconexão
           ( c_ucep[I] ? disreq ->
             ( c_mcep[I] ! unitreq(BuildDR) ->
               ( Disconnection_Requested := true ;
                 ( c_clock ! starttimer(P)  ->
                   CONNECTION_CLOSING ) ) ) )
           []
           - - pedido de conexão
           (c_mcep[I] ? unitind ->
            ((c_mcep[I] ! unitreq(BuildCC) ->
              DATA_TRANSFER)
             if Code(pdu) = CR else ERROR ) )
      )

- - esperando uma AK-pdu
WFAK == ( ( - - com o número de sequência correto
            c_mcep[I] ? unitind ->
```

```
                ( RESET_EMISSION_VARIABLES ;
                  E := (E + 1) mod 2;
                  ( c_clock ! stoptimer ->
                    DATA_TRANSFER ) )
                if (Code(pdu) = AK) ^ (Seq(pdu) = E) else
                ( - - com o número de sequência incorreto
                  ERROR
                  if (Code(pdu) = AK) ^ (Seq(pdu) <> E) else
                  WFAK ) )
          [] ( ( - - timeout ocorreu e AK não foi recebido
                 ( c_clock ? a_timeout ->
                   - - retransmite
                   c_mcep[I] ! unitreq(Emission_Buffer) ->
                   (Retrans_Attempts := Retrans_Attempts + 1;
                    WFAK ) ) )
               if  Retrans_Attempts < N  else  ERROR  )
          []   - - recepção de dados quando espera AK-pdu
               ( c_mcep[I] ? unitind -> R_ABRAC )
          []   - - requisição de desconexão q/ espera AK-pdu
               ( c_ucep[I] ? disreq ->
                 ( c_mcep[I] ! unitreq(BuildDR) ->
                   ( ( c_clock ! starttimer(P) ->
                       RESET_EMISSION_VARIABLES ) ;
                       ( Disconnection_Requested := true ;
                         CONNECTION_CLOSING ) ) ) )


R_ABRAC == ( ((Code(pdu) = DT) ^ (Seq(pdu) = R)) ->
               ( R := (R + 1) mod 2 ;
                 ( (( c_ucep[I] ! dataind(sdu)  ->
                      SKIP )  |||
                    ( c_mcep[I] ! unitreq(BuildAK(R)) ->
                      SKIP ) )
                  ; ( DATA_TRANSFER [] WFAK )  - - depende
                      de onde estava quando entrou em R_ABRAC
                 ) )
             [] ((Code(pdu)=DT) ^ (Seq(pdu) <> R))  ->
                  ( c_mcep[I] ! unitreq(BuildAK(R))  ->
                    ( DATA_TRANSFER [] WFAK ) )  - - depende
                      de onde estava quando entrou em R_ABRAC
             [] ( Code(pdu) = DR ) ->
                  ( c_ucep[I] ! disind ->
                    ( c_mcep[I] ! unitreq(BuildDC) ->
                      CONNECTION_OPENING ) ) )

- - Liberação de conexão
CONNECTION_CLOSING ==
        (   ( c_mcep[I] ? unitind ->
              ( INITIALIZE
                if((Code(pdu)=DC) v (Code(pdu)=DR)) else
                CONNECTION_CLOSING ) )
        []  (  - - reemissão de dr-pdu
               ( c_lock ? a_timeout ->
                 ( ( c_mcep[I] ! unitreq(BuildDR) ->
                     c_clock ! starttimer(P) ->
                     (Retrans_Attempts := Retrans_Attempts + 1;
                      Disconnection_Requested := true ;
                      CONNECTION_CLOSING ) ) )
                 if (Retrans_Attempts < N) else
                 CONNECTION_CLOSING

- - erros de protocolo
ERROR  ==  ( c_ucep[I] ! disind -> SKIP ) |||
             ( ( c_mcep[I] ! unitreq(BuildDR) ->
                 ( RESET_EMISSION_VARIABLES ;
                   ( Disconnection_Requested := true;
                     SKIP ) ) ) )
           ; CONNECTION_CLOSING )
```