LOTOS: uma técnica para a descrição formal de serviços e protocolos de comunicação (*)

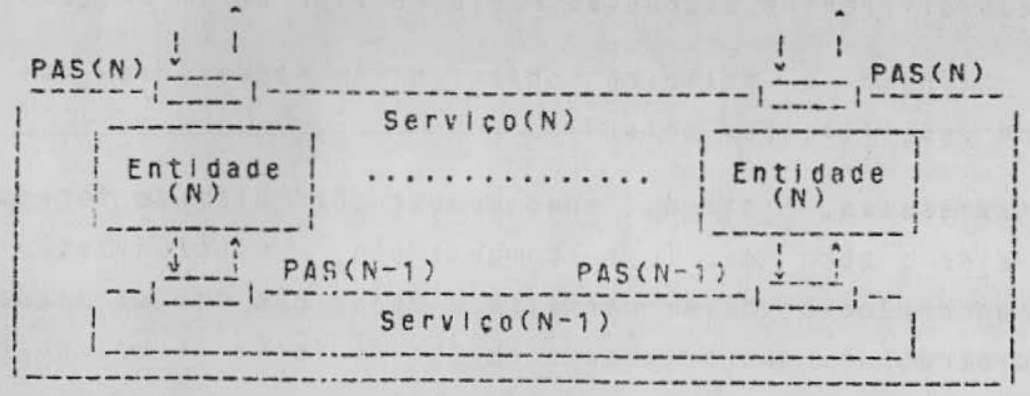
Wanderley Lopes de Souza
GRC/DSC/Universidade Federal da Paraíba

Sumário

Este artigo é uma introdução à técnica de descrição formal (TDF) "Language Of Temporal Ordering Specification (LOTOS)", que está sendo desenvolvida junto à "International Standard Organization (ISO). Inicialmente são apresentadas as caracaterísticas desejadas para uma TDF que vise a especificação de serviços e protocolos de comunicação. Em seguida são explicados os conceitos fundamentais relativos às origens de LOTOS e são introduzidos os operadores dessa linguagem. Finalmente é exemplificada a aplicação dessa técnica e são expostas as nossas conclusoēs.

1. Introdução

Uma rede de computadores, cuja arquitetura distribuída segue o Modelo Básico de Referência para a Interconexão de Sistemas Abertos (OSI) [ISO 83], é dividida em sete camadas funcionais. Utilizando os serviços oferecidos pela camada (N-1), as entidades da camada (N) cooperam entre si, de acordo com o protocolo (N), para fornecer um serviço (N) com mais recursos à camada (N+1).



(*) trabalho realizado com auxílio fornecido pelo CNPq

A noção de serviço (N) é abstrata pois condensa as camadas de 0 a N, omitindo o fluxo de dados entre elas. A especificação desse serviço é a descrição, por um observador externo, do comportamento de uma caixa preta, sujeita às trocas de primitivas de serviço com a camada superior, que são realizadas através dos pontos de acesso ao serviço (N) (PAS_N).

A especificação do protocolo (N) descreve o comportamento das entidades, que se comunicam, sincronizam-se e operam concorrentemente via os pontos de acesso ao serviço (N-1).

As descrições de seviços e protocolos tem sido realizadas associando-se uma linguagem natural a representações gráficas e/ou a tabelas de estados. As ambiguidades decorrentes dessas especificações semi-formais podem levar a implementações incompatíveis nas diferentes máquinas conectadas à rede. Além disso, erros provenientes da concepção e/ou especificação, que poderiam ser detectados e corrigidos nessas primeiras fases do desenvolvimento de um protocolo (ciclo de vida), podem proliferar por todas as implementações, tornando o trabalho de depuração de custo elevado e extremamente difícil. [BoNi 86]

Devido aos problemas acima mencionados, atualmente há uma tendência junto a ISO de complementar as especificações informais existentes e de desenvolver novos padrões, utilizando técnicas formais de descrição. Essas especificações formais devem:

(1) fornecer descrições claras e concisas do sistema que está sendo concebido e

(2) suportar uma análise rigorosa e validação passo-a-passo das diferentes etapas do ciclo de vida de um protocolo.

Para atingir o primeiro objetivo é necessário que a linguagem de especificação seja:

(a) expressiva, isto é, suas construções além de fornecer meios para exprimir comunicação, sincronização e concorrência, devem permitir a descrição dos serviços e protocolos das sete camadas OSI de forma hierárquica, modular e intuitiva e

(b) abstrata, no sentido de independência em relação aos métodos de implementação e no sentido de omissão, em qualquer etapa da especificação, dos detalhes irrelevantes.

Para atingir o segundo objetivo é necessário que a linguagem possua um grande poder analítico, ou seja, possua um modelo matemático que permita a verificação formal de propriedades desejadas para os objetos que estão sendo especificados. No caso do modelo de referência OSI, essa linguagem deve permitir a verificação de especificações, a validação de implementações e os testes de conformidade.

Infelizmente, quanto mais expressiva é uma linguagem mais difícil se torna a análise de seus programas. Essas duas características, conflitantes para a maioria das linguagens de programação, parecem ter encontrado um ponto de equilíbrio, no que diz respeito à técnica de descrição formal LOTOS.

2. Os conceitos e operadores fundamentais de LOTOS

LOTOS começou a ser desenvolvida por especialistas pertencentes ao grupo de trabalho ISO/TC97/SC21/WG16-1/FDT/ Subgroup C em 1981. Em LOTOS um sistema é especificado por um observador externo, que descreve os eventos incidentes nesse sistema e que modificam o seu comportamento, através da definição de uma relação temporal entre tais eventos. O modelo matemático global de LOTOS é dividido em duas partes distintas:

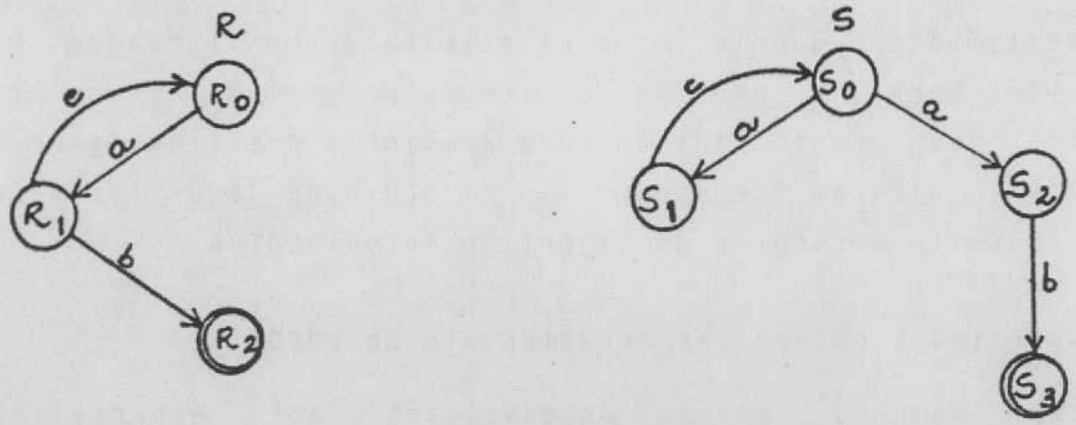
(1) um componente para a descrição das interações e dos comportamentos dos processos, que é uma extensão do "Calculus of Communicating Systems (CCS)", desenvolvido principalmente por Milner na University of Edinburgh [Mil 80] e que por sua vez é uma técnica analítica poderosa para a descrição de sistemas concorrentes e

(2) um componente para a descrição de estruturas de dados e de expressões, que é baseado na linguagem ACT ONE, desenvolvida pelo ACT-group da Technical University of Berlin [EhMa 85] e que por sua vez é uma álgebra para a especificação de tipos abstratos de dados.

Historicamente essas duas partes foram desenvolvidas separadamente e independentemente. Em princípio, outras linguagens que descrevem estruturas de dados podem ser combinadas ao primeiro modelo. Por esta razão e devido ao fato da parte dinâmica conter os conceitos básicos de LOTOS, nos limitaremos à exposição desse primeiro componente.

2.1. Observação externa do comportamento de processos

Para ilustrar esse princípio de base de LOTOS, utilizaremos dois agentes não determinísticos (R e S), definidos inicialmente através dos gráficos de transição



Questão: os agentes R e S são equivalentes ?

Segundo a equivalência tradicional dois gráficos são equivalentes, se eles aceitam a mesma linguagem (o mesmo conjunto de sequências).

$$R_0 = aR_1$$
$$R_1 = bR_2 + cR_0$$
$$R_2 = E \text{ (sequência vazia)}$$

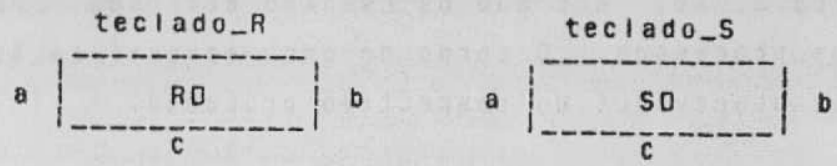
$$S_0 = aS_1 + aS_2$$
$$S_1 = cS_0$$
$$S_2 = bS_3$$
$$S_3 = E$$

Aplicando as propriedades de substituição e distribuição

$$R_0 = abE + acR_0 \qquad S_0 = abE + acS_0$$

e se $R_0 = S_0$ ==> R e S são equivalentes.

Os agentes R e S podem também ser representados por caixas pretas, contendo cada uma 03 teclas visíveis, e cujos comportamentos passaremos a investigar através de uma sucessão de experimentos atômicos

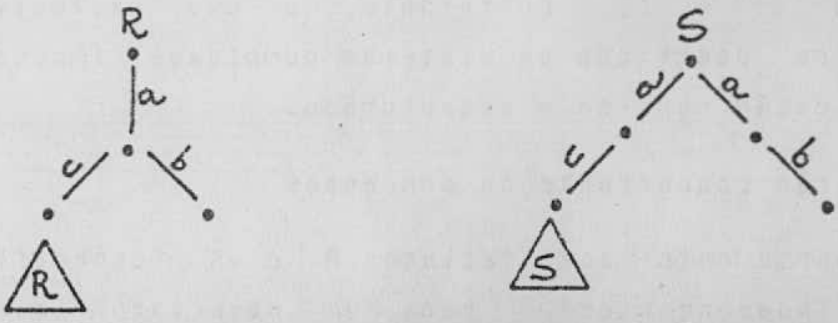


Experimento: apertar uma tecla tendo como resposta

(a) falha - tecla bloqueada) ou
(b) sucesso - tecla desbloqueada (uma transição secreta ocorre)

obs: em caso de ambiguidade, é escolhida uma transição de forma não-determinística (aleatória).

Após uma sucessão de experimentos, podemos traçar a seguinte árvore infinita, que representa melhor o comportamento dos dois agentes (caixas pretas), cujos estados internos são desconhecidos



Para um observador externo, após realizar com sucesso um experimento a no teclado_R ele sempre obterá sucesso se tentar o experimento b. No caso do teclado_S, ao repetir esse mesmo procedimento, algumas vezes ele terá sucesso e outras vezes ele encontrará a tecla b bloqueada (a ramificação à esquerda da árvore foi não-deterministicamente escolhida). Portanto, sob o ponto de vista de observação de comportamento, esses dois teclados não são equivalentes.

Esses teclados podem ser descritos em termos de processos, em LOTOS, por

```
process teclado_R [a,b,c] :=
  a ; ( b ; stop
      [] c ; teclado_R [a,b,c] )
endproc
```

```
process teclado_S [a,b,c] :=
      a ; b ; stop
   [] a ; c ; teclado_S [a,b,c]
endproc
```

onde teclado_R e teclado_S são os identificadores dos processos e os parâmetros a, b, e c são os eventos externos, passíveis de agir sobre os processos. O corpo de cada especificação define o comportamento observável do respectivo processo.

Nesses dois exemplos foram introduzidos o processo inativo (stop) , que pode representar uma situação de impasse, e as operações básicas ação (;) e escolha ([]) . A não equivalência entre os dois processos é refletida na não distributividade do operador ; em relação ao operador []. Cabe salientar ainda o uso de recursividade para a descrição dos comportamentos infinitos desses processos.

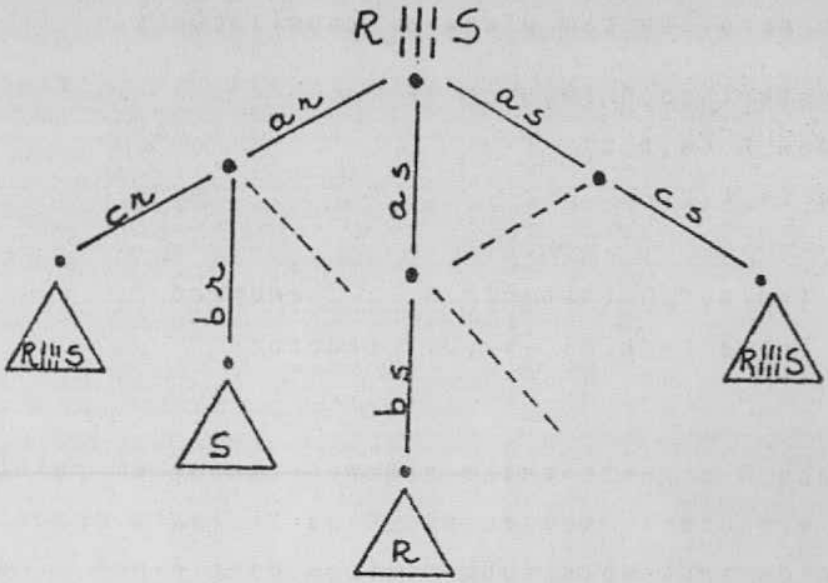
Em princípio, qualquer comportamento finito (ou infinito expresso através da recursividade) pode ser reduzido a uma sequência de ; e []. Entretanto, o uso exclusivo dessses operadores na descrição de sistemas complexos impossibilitaria uma especificação concisa e estruturada.

2.2. Composição concorrente de processos

O comportamento dos teclados R e S (especificados por processos independentes), para um observador que realiza sucessivos experimentos, escolhendo aleatoriamente um dos teclados a cada novo experimento, pode ser descrito, em LOTOS, através da composição paralela não-sincronizada ou entrelaçamento puro (|||)

```
process teclados [ar,br,cr,as,bs,cs] :=
      teclado_R [ar,br,cr]
  ||| teclado_S [as,bs,cs]
where
  process teclado_R [a,b,c] := .... endproc
  process teclado_S [a,b,c] := .... endproc
endproc
```

Mesmo que seja realizado um número finito de experimentos, a descrição de teclados, utilizando-se exclusivamente os operadores básicos, tornaria a especificação não concisa , dificultando a legibilidade. A árvore a seguir, que é uma transposição gráfica do comportamento observável de teclados, demonstra tais inconvenientes.



O nosso segundo conceito fundamental, é sincronização: comunicação entre dois processos que podem oferecer/aceitar experimentos complementares, que, em LOTOS, são eventos (ou portas) que compartilham o mesmo nome. (*)

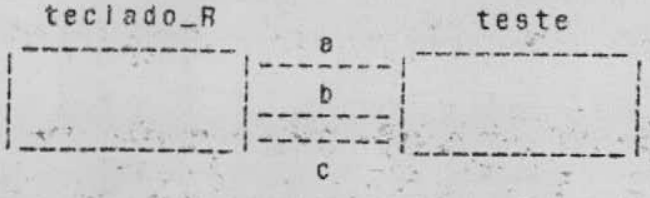
No experimento com os teclados, o sucesso de um experimento permite:

(I) ao teclado oferecer novos experimentos (se uma situação de impasse não foi atingida) e
(II) ao observador tentar novos experimentos.

O observador pode ser definido pelo processo

```
process teste [a,b,c] :=
     a : teste [a,b,c]
[] b ; teste [a,b,c]
[] c ; teste [a,b,c]
endproc
```

o sistema composto por teclado_R e teste pode ser representado



(*) em B1|||B2, se B1 e B2 possuem nomes iguais de portas ==> que eles podem se que comunicar com um ambiente comum e não entre si.

e o comportamento resultante pode ser descrito através da composição paralela com plena sincronização (||)

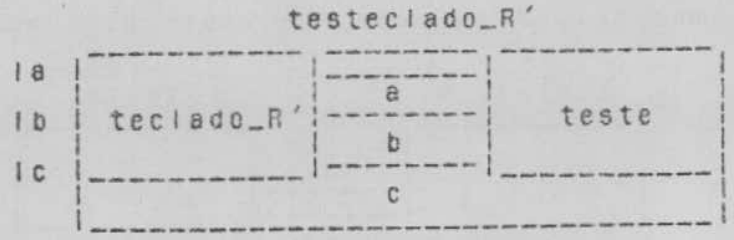
```
process testeclado_R [a,b,c] :=
     teclado_R [a,b,c]
  || teste [a,b,c]
where
  process teclado_R [a,b,c] := .... endproc
  process teste [a,b,c] := .... endproc
endproc
```

onde teclado_R e teste devem sincronizar-se em relação a todos os eventos e o processo testeclado_R participará somente dos eventos nos quais os seus dois subprocessos participem. Há um dualismo entre as propriedades relativas ao operador || e as relativas ao operador |||.

Vamos supor que desejássemos automatizar o procedimento de teste, ou seja, combinar teclado_R com teste e analisar como esse sistema composto se comportaria para um novo observador. Para facilitar essa nova observação associaremos a cada tecla um sinalizador ("led"), que indicará (acenderá) ao observador a ocorrência dos sucessos dos experimentos relativos às respectivas teclas. Eliminaremos também a situação de impasse, que existe na definição do teclado e redefiniremos o seu comportamento para

```
process teclado_R' [a,la,b,lb,c,lc] :=
     a ; la ; teclado_R' [a,la,b,lb,c,lc]
  [] b ; lb ; teclado_R' [a,la,b,lb,c,lc]
  [] c ; lc ; teclado _R' [a,la,b,lb,c,lc]
endproc
```

o sistema composto por teclado_R' e teste pode ser representado



e o comportamento resultante pode ser descrito através da composição paralela com sincronização nos eventos a, b, c (|[a,b,c]|)

```
process testeclado_R' [a,la,b,lb,c,lc] :=
            teclado_R' [a,la,b,lb,c,lc]
  |[a,b,c]| teste [a,b,c]
where
  process teclado_R' [a,la,b,lb,c,lc] := .... endproc
  process teste [a,b,c] := .... endproc
endproc
```

Nesse exemplo a lista dos eventos para sincronização é explícita e podemos concluir que ||| e || são casos particulares de |[a,b,c,...]|, onde a lista dos eventos para sincronização é vazia para o primeito operador e o alfabeto completo para o segundo.

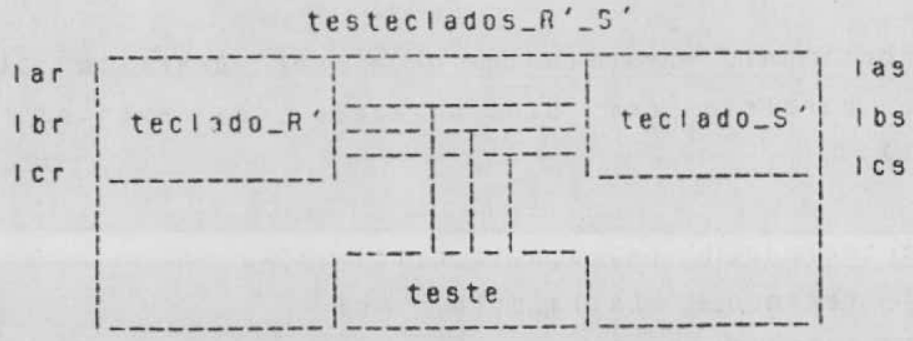
Para o nosso novo observador essa composição não corresponde à realidade, já que os testes são realizados automaticamente e não lhe é permitido o acesso aos eventos a, b e c. Portanto, é necessário esconde-los, o que poderia ser feito pelo próprio operador |[a,b,c]|.

A solução apresentada acima não seria conveniente no caso da composição de mais de dois processos, pois a operação |[a,b,...]| não seria associativa. No nosso exemplo, esse fato pode ser verificado para o caso do compartilhamento do teste pelos teclado_R' e teclado_S' (onde esse último processo é definido da mesma forma que o anterior).

(teclado_R'[..] |[a,b,c]| teste[..]) |[a,b,c]| teclado_S'[..] ≠ teclado_S'[..] |[a,b,c]| (teste[..] |[a,b,c]| teclado_R'[..])

A solução adotada foi introduzir um segundo operador para ocultar os eventos internos. Resultado: |[..]| compoẽ e hide..in esconde.

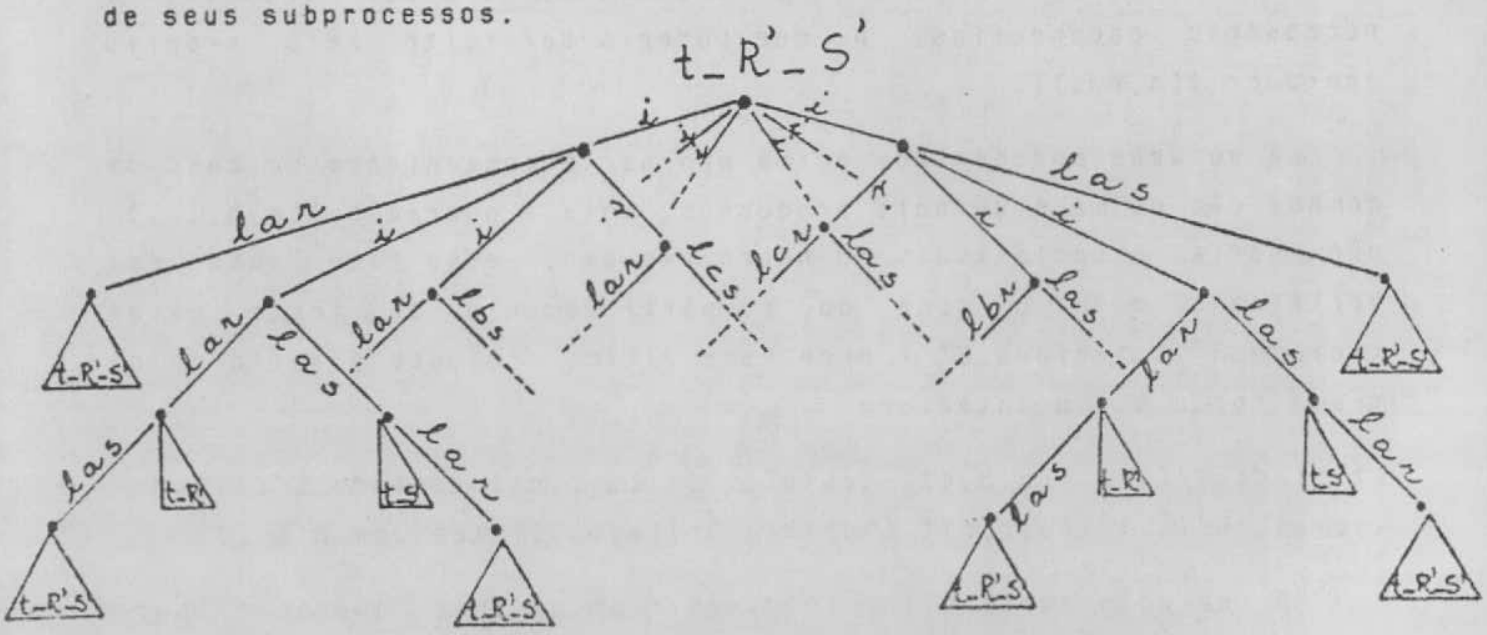
A combinação desses dois operadores para a composição sincronizada dos três processos acima, pode ser representada por



o processo resultante dessa composição é

```
process testeclados_R'_S' [lar,lbr,lcr,las,lbs,lcs] :=
  hide a,b,c in
              teclado_R' [a,lar,b,lbr,c,lcr]
    |[a,b,c]| teste [a,b,c]
    |[a,b,c]| teclado_S' [a,las,b,lbs,c,lcs]
where
  process teclado_R' [a,la,b,lb,c,lc] := .... endproc
  process teclado_S' [a,la,b,lb,c,lc] := .... endproc
  process teste [a,b,c] := .... endproc
endproc
```

e o comportamento de testeclados_R'_S' pode ser traçado através de uma árvore, que é uma combinção das árvores de comportamento de seus subprocessos.



Na árvore estão descritas as possibilidades iniciais de evolução do processo testeclados_R'_S', onde I revela a ocorrência de um evento interno. Nessa árvore, uma sequência de dois eventos I indica a possibilidade de duas teclas (de teclados diferentes) serem testadas consecutivamente com sucesso, antes que ocorra a sinalização da primeira tecla (velocidade de teste > velocidade de sinalização). Há casos, como no exemplo vending-machine utilizado em [Brink 86], em que os efeitos dos eventos internos no comportamento do sistema, podem ser percebidos pelo observador. Esse fato vem reforçar as justificativas para a representação de tais eventos.

2.3. Desabilitação e composição sequencial de processos

Frequentemente deseja-se expressar a possibilidade de eventos (internos ou externos) frustar ou desativar o curso normal de execução de um processo. Por exemplo, na composição teclado_R' |[..]| teste o nosso observador pode desejar controlar o início e a reinicialização do sistema. Essa alternativa está incorporada no processo teste' e a composição resultante é

```
process teste'teclado_R' [on,la,lb,lc] :=
  ( hide a,b,c in
      teclado_R' [a,la,b,lb,c,lc] |[a,b,c]| teste' [on,a,b,c] )
  [> teste'teclado_R' [on,la,lb,lc]
where
  process teclado_R' [a,la,b,lb,c,lc] := .... endproc
  process teste' [on,a,b,c] :=
    on ; teste [a,b,c]
  where
    process teste [a,b,c] := .... endproc
  endproc
endproc
```

A operação de desabilitação B_1 [> B_2 define um processo, no qual a execução de B_1 (teclado_R' |..| teste') pode ser interrompida a qualquer instante pela ocorrência do evento inicial on de B_2 (teste'teclado_R'). Caso o evento inicial de B_2 ocorra antes do evento inicial de B_1, somente B_2 será executado e caso B_1 tenha terminado a sua execução com sucesso, B_2 não poderá

mais interrompê-lo. Essas duas últimas situações não estão bem esclarecidas no exemplo acima, devido ao uso de recursividade.

Frequentemente deseja-se também expressar sequencialidade entre processos (e não entre eventos), de forma a refletir a estrutura do sitema na estrutura da especificação. Por exemplo, vamos supor que o nosso observador deseje realizar experimentos, alternando a cada novo experimento o teclado. Essa situação, conjuntamente com a redefinição dos subprocessos envolvidos, é descrita por

```
process teste"tecado_R"_S" [on,lar,lbr,lcr,las,lbs,lcs] :=
     ( hide a,b,c in
                    teclado_R" [a,lar,b,lbr,c,lcr]
          |[a,b,c]| teste [on,a,b,c] )
  >> ( hide a,b,c in
                    teclado_S" [a,las,b,lbs,c,lcs]
          |[a,b,c]| teste [on,a,b,c] )
  >> teste"teclado_R"_S" [on,lar,lbr,lcr,las,lbs,lcs]
where
  process teclado_R" [a,la,b,lb,c,lc] :=
       a ; la ; exit
    [] b ; lb ; exit
    [] c ; lc ; exit
  endproc
  process teclado_S" [a,la,b,lb,c,lc] := .... endproc
  process teste" [on,a,b,c] :=
    on ; ( a ; exit
         [] b ; exit
         [] c ; exit )
  endproc
endproc
```

O término com sucesso de B_1 (teclado_R" |[..]| teste"), isto é, o término normal (indicado pelo processo básico exit), não devido a um impasse, habilita (>>) a execução de B_2 (teclado_S" |[..]| teste"), cujo término bem sucedido, por sua vez, relança todo o sistema. No caso da composição de processos (como no exemplo acima), para que haja um término bem sucedido da composição é necessário que todos os processos envolvidos terminem com sucesso.

3. Interações estruturadas entre processos

Valores e estruturas de dados são descritos em LOTOS empregando-se, atualmente, a linguagem para especificação de tipos abstratos de dados ACT ONE. Neste artigo não discutiremos as definições dos tipos propriamente ditas, aplicando-os, entretanto, nas interações estruturadas entre processos.

Em CCS a passagem de valores numa comunicação entre dois processos baseia-se na possibilidade desses processos oferecer/aceitar tais valores através de portas complementares. LOTOS aproveita e estende esse princípio, utilizando uma notação que é proveniente de "Communication Sequencial Processes (CSP)" [BrHoRo 84], que descreveremos a seguir:

a?x:t — o processo está preparado para aceitar um valor do tipo t na porta a. Após a ocorrência desse evento, x assume o valor aceito e

a!E — o processo está preparado para oferecer o valor E, que pode ser uma expressão arbitrária (e.g., 2*5+3), na porta a.

O procedimento descrito acima, pode ser generalizado para o caso de múltiplas ofertas/aceitações. Por exemplo, um processo pode aceitar na porta a

a ? x:Int ? y:char ? z:bool

e se comunicar com um processo que ofereça na porta a

a ! 5 ! $! true

mas não pode se comunicar com um processo que ofereça

a ! 3.2 ! $! true

Como já dissemos, LOTOS amplia esse princípio de forma a permitir que dois processos possam interagir nos casos abaixo

processo A	processo B	condição para a interação	tipo de interação	consequência da interação
a?x:t	a!E	valor de E no domínio de t	passagem de valor	após a interação x assume o valor de E
a!E1	a!E2	valor de E1 = valor de E2	casamento de valores	sincronização pura
a?x:t	a?y:u	t = u	negociação ou geração de valores	após a interação x=y= um valor no domínio (t=u)

Assim sendo, as possibilidades de interação de um processo são definidas pelos nomes de suas portas (ou eventos), cada porta acompanhada de uma lista de atributos. Um processo B_1 que possua uma porta a onde

```
a ? x:Int ! true ? y:char
```

pode interagir com um processo B_2 em

```
a ! (2*5) ! true ? z:char
```

mas não pode interagir com esse mesmo processo em

```
a ! (2*5) ! false ? z:char
```

3.1. Parametrização de processos

Como já foi visto nos vários exemplos anteriores, LOTOS permite que um processo seja identificado de forma abstrata pelo seu nome, de tal forma que nas ocorrências de uma abstração os nomes formais das portas são substituídos pelos nomes verdadeiros. LOTOS permite também a inclusão de parâmetros na definição de um processo, de tal forma que nas ocorrências de sua abstração, as variáveis definidas possam ser substituídas por expressões de valor do mesmo tipo (análogo às construções abstratas de funções e procedimentos nas linguagens de programação). Por exemplo, são permitidas expressões do tipo

```
process compare [in,out,out'] (max,min:int) :=
  in?x:int ; ( [min < x < max] --> out!x ; exit
             [] [ x <= min] --> out'!min ; exit
             [] [ x >= max] --> out'!max ; exit
endproc
```

onde compare [a,b,c] (10,-10) é uma possível ocorrência da abstração compare do processo acima definido. Cabe salientar ainda, a presença do operador condicionador [..] -->, que combinado a [] pode condicionar uma série de alternativas (uma espécie de case).

Parâmetros são também utilizados para generalizar a operação de escolha ([]), produzindo um novo operador

$$\text{choice } x_1:t_1,\ldots,x_n:t_n \ [] \ B(x_1,\ldots,x_n)$$

onde $x_1, \ldots, x_n$ são variáveis indexadoras. Por exemplo, vamos supor que a expressão de comportamento B(x) dependa da variável x do tipo inteiro. Podemos especificar a escolha entre os processos B(E) para todos os valores inteiros através da expressão

```
choice x:int [] B(x)
```

Indentificadores de portas também são usados como indexadores, na operação

$$\text{choice } g \text{ in } [g_1, \ldots, g_n] \ [] \ B$$

que permite a escolha entre qualquer um dos processos $B[g_j/g]$ para $(1 <= j <= n)$, onde / é a re-rotulação de g em g_j.

As demais operações em LOTOS referem-se à declaração de variáveis locais

$$\text{let } x_1:t_1=E_1, \ldots, x_n:t_n=E_n \text{ in } B$$

e à inicialização de processos

$$p[g_1, \ldots, g_n] (E_1, \ldots, E_n)$$

No Anexo 1 é apresentado um resumo dos operadores LOTOS. Eles estão organizados hierarquicamente em termos de prioridade decrescente, a fim de que seja evitado o uso excessivo de parêntesis.

3.2. Funcionalidade dos processos

A fim de permitir a passagem de valores entre processos, uma lista finita de expressões de valor pode ser adicionada ao processo exit, resultando em

$$\text{exit } (E_1, \ldots, E_n) \text{ ou}$$
$$\text{exit (any t)}$$

onde $E_1, \ldots, E_n$ são as expressões de valor a serem passadas ao processo subseqüente e any indica que qualquer expressão de valor do tipo t pode ser passada ao processo seguinte.

A fim de estabelecer as regras que irão reger a passagem de valores entre processos, é introduzido o conceito de funcionalidade.

A funcionalidade de um processo é o produto cartesiano dos domínios dos valores que são passados, quando do término desse processo. Na tabela abaixo são apresentadas as funcionalidades, para os casos em que elas podem ser bem definidas, das expressões de comportamento LOTOS.

expressões de comportamento	condições	funcionalidades bem definidas
stop	inativo	noexit
exit	sem passagem de valor	exit
exit (E1,..,En)	E1:t1,..,En:tn	func(t1) x..x func(tn)
B = B1 [] B2 ou = B1 [> B2	se func(B1)=func(B2) se func(B1)=noexit se func(B2)=noexit	func(B1) func(B2) func(B1)
B = choice..[] B'		func(B')
B = B1|||B2 ou = B1||B2 ou = B1|g1..gn|B2	se func(B1)=func(B2) se func(B1)=noexit se func(B2)=noexit	func(B1) noexit noexit

Para exemplificar, vamos considerar as seguintes expressões de comportamento

B_1 = a?x:int ?y:bool ; b!x !y ; exit(x,y)
B_2 = a?x:int ; ([x >= y] --> b!x ; stop [] [x < y]->b!y ; stop)
B_3 = (a?x:int ; b!x ; exit(x)) |[b]| (b?y:int ; c!y ; exit(y))
B_4 = (a?x:int ; b!x ; exit(x)) ||| (b?y:int ; c!y ; stop)
B_5 = (a?x:int ; b!x ; exit(x)) [> (b?y:bool ; c!y ; exit(y))

que possuem as seguintes funcionalidades

func(B_1) = dominio(int) x dominio(bool), func(B_2) = noexit,
func(B_3) = dominio(int), func(B4) = exit e func(B_5) é indefinida

Se a funcionalidade da expressão de comportamento resultante de um processo é bem definida, ela pode ser incluida na lista dos parâmetros formais na definição abstrata desse processo.

Por exemplo, se B_1, B_2, B_3 e B_4 são as expressões de comportamento resultantes dos seus respectivos processos, as definições abstratas de tais processos teriam a forma

process B_1 [a,b] : int, bool := .... endproc
process B_2 [a,b] (y:int) : noexit := .... endproc
process B_3 [a,b,c] : int := .... endproc
process B_4 [a,b,c] : exit := .... endproc

Uma vez definidas a parametrização e a funcionalidade de processos, podemos generalizar a composição sequencial a fim de permitir a passagem de informações entre os processos envolvidos nessa operação

$$B_1 \gg \text{accept } x_1 : t_1, \ldots, x_n : t_n \text{ in } B_2$$

onde func(B_1) = dominio(t_1) x....x dominio(t_n) e x_1 .... x_n são os nomes das variáveis usadas em B_2 que assumirão os valores passados por B_1, quando do seu término bem sucedido.

Na especificação do serviço de transporte, durante a fase de estabelecimento da conexão, são negociados o envio de dados urgentes e a qualidade do serviço. A passagem dessas informações para a fase de transferência de dados pode ser definida abstratamente como

```
   fase_de_conexão [tsapA,tsapB] (....)
>> accept qualidade_de_serviço : quality_parameter_sort
          opção_dados_urgentes : boolean
   in fase_de_dados [tsapA,tsapB] ( qualidade_de_serviço,
                                    opção_dados_urgentes )
```

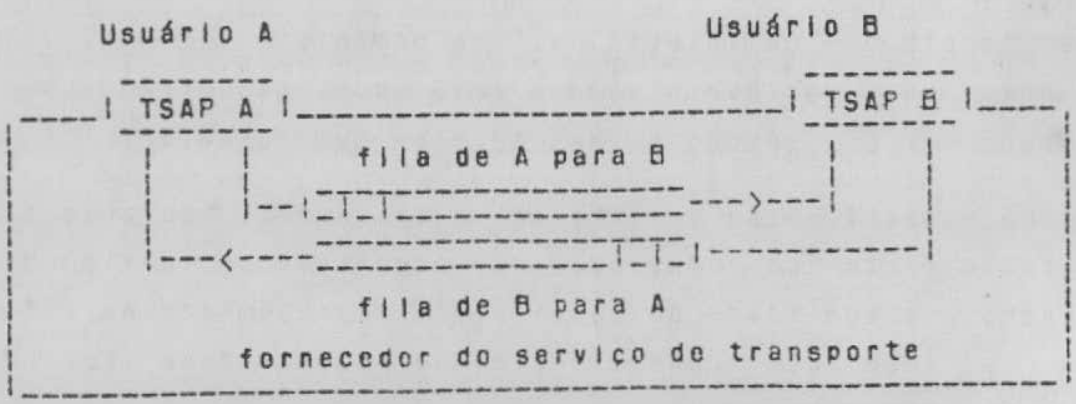
4. Um exemplo de especificação

Uma especificação em LOTOS é sintaticamente estruturada da seguinte forma:

```
specification<identificador>[[lista de portas]]:<funcionalidade>
:=
<expressão de comportamento>
where
  <definições de tipos>
  <definições de processos>
endspec
```

Em [Brka 84] é especificado um serviço de transporte restrito, que é basicamente constituido de um processo TS, que descreve o comportamento do serviço de transporte em termos das primitivas que podem ser observadas e, portanto, trocadas com os usuários desse serviço. Esse processo é constituído de vários sub-processos, entre os quais existem aqueles referentes às diferentes fases ("estados") da estação de transporte (WAIT_RESPONSE, WAIT_CONFIRMATION, ...). Dentre essas fases estamos interessados particularmente na transferência de dados (process DATA_TRANSFER).

Esse serviço de transporte pode ser representado por



cujo comportamento é modelado pelos dois pontos de acesso ao serviço de transporte (TSAP A e TSAP B) e por duas filas unidirecionais que obedecem à disciplina FIFO para os dados normais, mas que permitem aos dados urgentes ultrapassar os normais.

A expressão de comportamento da fase de transferência de dados, pode ser descrita por um processo LOTOS parametrizado por um tipo abstrato de dado (exp_queue) que modela o conteúdo das filas de transmissão.

A definição do tipo exp_queue, que é feita de acordo com a sintaxe e a semântica de ACT ONE, deve corter basicamente as operações que permitem as manipulações com a fila e com seus elementos e as equações que definem as relações entre as operações.

No caso do processo DATA_TRANSFER , que é apresentado no Anexo 2, s_1 e s_2 são os parâmetros que representam as filas de transmissão e edo indica a escolha ou não da opção dados urgentes (a e b representam os TSAPs). O princípio de operação dessas filas é bastante simples: utilizando-se as diferentes operaçõcs definidas em exp_queue, verifica-se se uma fila contem dados urgentes. Em caso afirmativo, o primeiro elemento da fila que contem esse tipo de dado é oferecido ao usuário receptor através da primitiva EXDATInd. Caso contrário o primeiro elemento da fila é oferecido ao usuário através da primitiva DATInd (obviamente que o usuário transmissor envia dados urgentes através de EXDAT_req e dados normais através de DAT_req)

É interessante comparar a especificação do Anexo 2 com a que foi realizada em [ISO 86] , sob o nome de Expedited_data_queue, pois ambas descrevem o mesmo comportamento. Nessa última não foi utilizado um tipo abstrato de dado para as filas, empregando-se somente os parâmetros nrm_data e exp_data, trocando dessa forma a complexidade da descrição de um tipo abstrato de dado pela complexidade da descrição das interações do processo.

5. Conclusão

Neste trabalho procuramos apresentar os conceitos e as construções da técnica de descrição formal LOTOS, procurando demonstrar, através de exemplos simples e reduzidos, as capacidades de expressão e abstrção dessa linguagem.

Restringimo-nos ao componente dinâmico da linguagem, que é a essência de LOTOS, mas acreditamos que um estudo aprofundado do componente que define os tipos de LOTOS se faz necessário, principalmente se quisermos utilizar essa linguagem para fins de validação.

A semântica formal de LOTOS, que é baseada num conjunto de regras de inferência e o conceito formal (axiomas e teoremas) de equivalência entre comportamentos de processos, serão abordados num trabalho futuro, quando investigaremos a capacidade de análise dessa linguagem.

6. Referências

[BoNi 86] T. Bolognesi, R. Nicola, "A tutorial on LOTOS", submetido a Computer Networks.

[BrHoRo 84] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, "A Theory of Communicating Sequencial Processes", JACM, Vol. 31, No. 3, Jul/84, pp. 560-599.

[Brink 86] E. Brinksma, "A tutorial on LOTOS", Protocol Specification, Testing and Verification, V, editado por M. Diaz, North-Holland, 1986, pp. 171-194.

[BrKa 84] E. Brinksma, G. Karjoth, "A specification of the OSI transport service in LOTOS", Protocol Specification, Testing and Verification, IV, editado por Y. Yemini, R. Strom and S. Yemini, North-Holland, 1985, pp. 227-251.

[EhMa 85] H. Ehrig, B. Mhar, " Fundamentals of Algebraic Specification 1 _ Equations and Initial Semantics", Spring-Verlag, 1985.

[ISO 83] ISO IS 7498, "Information Processing Systems _ Basic Reference Model for Open Systems Interconnection", 1986.

[ISO 86] ISO DP 8807, "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", março/86.

[Mil 80] R. Milner, "A Calculus of Communicating Systems", editado por G. Goos e J. Hartmanis, Spring-Verlag, 1985.

Anexo 1

Sintaxe dos operadores LOTOS

nome	sintaxe concreta
hinding	hide g1,..,gn in B
action-prefix	g;B, g?x:t;B, g!E;B, i;B (formatos básicos) g d1 .... dn [BE] ; B (formato geral)
guarding	[BE] --> B
choice	B1 [] B2
parallel composition	B1 |[g1,...,gn]| B2 (formato geral) B1 ||| B2 (entrelaçamento puro) B1 || B2 (sincronização plena)
enabling	B1 >> B2, B1 >> accept x:t1,...,xn:tn in B2
disabling	B1 [> B2
summation	choice g in [g1,...,gn] [] B (em portas) choice x:t [] B (em valores)
local-definition	let x1:t1=E1,...,xn:tn=En in B
instantiation	P[g1,...,gn] (E1,...,Em)
termination	exit, exit (E1,...,En)
inaction	stop

Na tabela acima a seguinte convenção foi adotada:

(a) B, B1, B2 representam expressões de comportamento;

(b) E1, ..., Em, En representam expressões de valor e BE uma expressão booleana;

(c) g, g1, ..., gn representam identificadores de portas;

(d) t, t1, ..., tn representam identificadores de tipos

(e) x, x1, ..., xn representam identificadores de valores e

(f) d1, ..., dn representam eventos.

Anexo 2

Descrição do processo de transferência de dados no contexto da especificação do serviço de transporte

```
specification (* serviço de transporte *)

def (* definições ACT ONE *)
-------------------
def exp_queue is element with bool0 with

  sort: queue

  opns: new: --> queue
        add: el, queue --> queue
        add_exp: el, queue --> queue
        rest: queue --> queue
        rest_exp: queue --> queue
        first: queue --> element
        first_exp: queue --> element
        is_empty: queue --> bool
        is_empty_exp: queue --> bool
        if then else: bool, queue, queue --> queue

    equs: rest(new) = new
        rest(add(x,q)) = if is_empty(q)
                         then new
                         else add(x, rest(q))

        rest(add_exp(x,q)) = if is_empty(q)
                             then new
                             else add_exp(x, rest(q))

        rest_exp(new) = new
        rest_exp(add(x,q)) = add (x, rest_exp(q))

        rest_exp(add_exp(x,q)) = if is_empty_exp(q)
                                 then q
                                 else add_exp(x, rest_exp(q))

        first(new) = error
        first(add(x,q)) = if is_empty(q)
                          then x
                          else first(q)
        first(add_exp(x,q)) = if is_empty(q)
                              then x
                              else first(q)

        first_exp(new) = error
        first_exp(add(x,q)) = if is empty(q)
                              then error
                              else first_exp(q)
        first_exp(add_exp(x,q)) = if is empty_exp(q)
                                  then x
                                  else first_exp(q)

        is_empty(new) = true
        is_empty(add(x,q)) = false
        is_empty(add_exp(x,q)) = false
        is_empty_exp(new) = true
        is_empty_exp(add(x,q)) = is_empty_exp(q)
        is_empty_exp(add_exp(x,q) = false
        if true then q1 else q2 = q1
        if false then q1 else q2 = q2

end of def


process TS[a,b] :=

---------------------
```

```
where

process.................

process DATA_TRANSFER[a,b](s1,s2:exp_queue,edo:boolD) :=

   [not(is_empty_exp(s1))] --> EXDATInd[b](first_exp(s1))
              ;DATA_TRANSFER[a,b](rest_exp(s1),s2,edo)

[] [not(is_empty_exp(s2))] --> EXDATInd[a](first_exp(s2))
              ;DATA_TRANSFER[a,b](s1,rest_exp(s2),edo)

(*  se  uma fila contem ExDATA,  então o primeiro elemento  desse
tipo deve ser removido e entregue ao usuário receptor *)

[] [first(s1) <> error] --> DATInd[b](first(s1))
               ;DATA_TRANSFER(rest(s1),s2,edo)

[] [first(s2) <> error] --> DATInd[a](first(s2))
               ;DATA_TRANSFER(s1,rest(s2),edo)

(*  se o primeiro elemento da fila é um dado normal ele deve  ser
removido e entregue ao usuário receptor *)

[] [edo] --> ( EXDATreq[a](x);DATA_TRANSFER(add_exp(x,s1),s2,edo)
          [] EXDATreq[b](x);DATA_TRANSFER(s1,add_exp(x,s2),edo)

(* se a opção para a transferência de dados urgentes é  escolhida
então os elementos ExDATA são aceitos *)

[] DATreq[a](x);DATA_TRANSFER(add(x,s1),s2,edo)

(* aceita dado normal do lado do usuário que chama. Neste exemplo
somente o usuário a pode solicitar uma conexão *)

[] DATreq[b](x);DATA_TRANSFER(s1,add(x,s2),edo)

(* aceita dado normal do lado do usuário chamado,  que no caso só
pode ser b *)

[] TERMINATION[a,b](s1,s2,edo)

(* processo que descreve todos os casos de desconexão *)

end
-----------------------
endspec
```