SIMULAÇÃO DE ESPECIFICAÇÕES LOTOS USANDO LINGUAGENS FUNCIONAIS

Carlos A. G. Ferraz
Paulo R. F. Cunha
Silvio R. L. Meira

Departamento de Informática
Universidade Federal de Pernambuco
CP 7851, 50739, Recife-PE, Brasil

RESUMO

O objetivo deste trabalho é mostrar como podemos usar a técnica de prototipação rápida para tornar executáveis especificações descritas em LOTOS. Para tanto, usamos linguagens de programação funcionais para escrever os protótipos e simular as expressões LOTOS. A simulação é mostrada através de exemplos, onde são explicados os procedimentos de tradução das especificações para os protótipos funcionais.

ABSTRACT

The objective of this document is to show how we can use the rapid prototyping technique to make executable specifications written in LOTOS. We use functional programming languages to write the prototypes and simulate the expressions of LOTOS. The simulation is showed through examples, where the procedures of translation from the specifications to the functional prototypes are explained.

1. INTRODUÇÃO

LOTOS (*Language Of Temporal Ordering Specification*) [BoB87,

ISO88] é uma das duas Técnicas de Descrição Formal desenvolvidas pela ISO (ISO/TC97/SC21/WG1/FDT/Subgroup C) para a especificação de sistemas distribuídos, em especial aqueles que seguem a arquitetura OSI (*Open Systems Interconnection*). Embora o nome sugira relação com lógica temporal, a técnica se baseia em álgebra de processos. Assim sendo, LOTOS é uma técnica algébrica de especificação formal, que permite definições claras, não-ambíguas e independentes de implementação.

A idéia era torná-la executável. No entanto, esta característica parece não ter sido, ainda, explorada o suficiente para tal. O caminho seguido por aqueles que esperam ver as especificações executadas em computador tem sido o que parece ser mais rápido, que é o de simulação usando linguagens de programação cujas características se assemelhem às de LOTOS. A maior parte destes trabalhos têm se baseado em linguagens de programação lógica do tipo Prolog [Gil87,GHL88] ou em sistemas de transição [Kar88], por causa de sua relação com a semântica operacional de LOTOS. Nossa proposta é usar programação funcional para fazer prototipação rápida a fim de simular as especificações descritas em LOTOS. Aqui os protótipos são descritos apenas em relação à parte dinâmica de LOTOS, ou seja, o componente que descreve as interações e comportamentos de processos e cujas idéias baseiam-se, principalmente, no Cálculo de Sistemas Comunicantes (CCS) de Milner [Mil80]. Um segundo componente é o que descreve as estruturas de dados e expressões de valores, e baseia-se na técnica de tipos abstratos de dados ACT ONE [EhM85]. Nossas idéias acerca da especificação funcional de tipos de dados que correspondam a este componente estão em [FCM89].

2. LOTOS: ASPECTOS GERAIS

A idéia básica de LOTOS é que sistemas sejam especificados definindo-se a relação temporal entre as interações que formam

o comportamento externamente observável de um sistema. Este é descrito em termos de processos, que realizam eventos internos e se comunicam via eventos de comunicação. Assim, a especificação de um sistema é, essencialmente, uma hierarquia de definições de processos que se comunicam.

A interação entre processos ocorre se ambos têm habilitados os mesmos eventos. Uma oferta de evento consiste de um *ponto de interação* ou *porta* e uma lista finita (não vazia) de *atributos*. Dois tipos de atributos são possíveis: *declaração de valor* (p!value-expr) e *declaração de variável* (p?var:type). A sincronização ocorre se os atributos respeitam as condições de sincronização da Tabela 1. Por exemplo, se dois processos que estão executando em paralelo oferecem, respectivamente, os seguintes eventos:

p ?x:int !'LOTOS' !true e
p !7 ?text:string !true

após a interação x = 7, text = 'LOTOS' e "!true" e "!true" apenas sincronizam os processos.

Processo A	Processo B	Condição de sincron.	Tipo de interação	Efeito
g ! E1	g ! E2	valor(E1) = valor(E2)	casamento de valor	sincronização
g ! E	g ? x:t	valor(E) ∈ domínio(t)	passagem de valor	depois da sinc x = valor(E)
g ? x:t	g ? y:u	t = u	geração de valor	depois da sinc x = y = v para v um valor ∈ domínio(t)

Tabela 1 - Tipos de Interacão

O comportamento de um processo em LOTOS é descrito por expressões de comportamento, que podem ser formadas através

dos seguintes operadores principais:

- **sequencialidade**: a expressão a;B determina que após a ocorrência do evento a a expressão de comportamento B pode ser executada.
- **escolha**: B1 [] B2 indica que B1 ou B2 serão escolhidos de acordo com um dos critérios abaixo:
  a) não-determinístico, isto é, o ambiente oferece um evento que possibilita que mais de uma expressão seja escolhida. Assim, a escolha de apenas uma delas será aleatória;
  b) se apenas uma das alternativas contém o evento ofertado como seu evento inicial, então ela será escolhida;
- **paralelismo**: B1 ||| B2 determina execução concorrente *sem sincronização*; B1 || B2 indica execução concorrente *com sincronização plena*; B1 |[$a_1,\ldots,a_n$]| B2 representa execução concorrente *relativa aos eventos* $a_1,\ldots,a_n$. Este último é chamado *operador geral* de paralelismo, pois serve também para simular os dois anteriores, considerando uma lista vazia ([]) para o primeiro (|||), e uma lista com todos os eventos comuns aos processos com plena sincronização (||).
- **composição sequencial**: B1 >> B2 expressa que após a terminação com sucesso de B1, B2 pode executar.
- **desabilitação**: B1 [> B2 indica que se o evento inicial de B2 ocorrer antes que B1 termine, B1 deixará de ser executado, em favor de B2.
- **hide**: hide $a_1,\ldots,a_n$ in B determina que os eventos $a_1,\ldots,a_n$ serão usados apenas para a comunicação interna entre os processos que estão executando em paralelo na expressão B, ou seja, os eventos serão *escondidos* do observador externo.

A sintaxe das principais expressões LOTOS é mostrada na Tabela 2 a seguir.

NOME	SINTAXE
Inação	stop
Terminação com sucesso	exit
Ação - não-observável (interna) - observável	 i; B a; B
Escolha	B1 [] B2
Paralelismo - sem sincronização - sincronização plena - caso geral	 B1 ||| B2 B1 || B2 B1 |[$a_1,\ldots,a_n$]| B2
Composição sequencial	B1 >> B2
Desabilitação	B1 [> B2
Hiding	hide $a_1,\ldots,a_n$ in B
Instanciação	p [$a_1,\ldots,a_n$]

Tabela 2 - Sintaxe de LOTOS básico

3. PROGRAMAÇÃO FUNCIONAL

Na programação funcional, os objetos sendo construídos são mais importantes do que o método de construção, ou seja, o QUÊ importa mais do que o COMO. Este estilo *denotacional* permite a construção direta de *protótipos*, bem como a geração de *especificações executáveis*, possibilitando validação de projetos.

Funções são, na maior parte das linguagens de programação funcionais, *objetos de primeira classe*, isto é, podem ser passadas como argumentos, retornadas como resultado e armazenadas em variáveis. Esta característica nos permite definir os processos de LOTOS como funções, para passá-los, por exemplo, como argumentos para funções que executam expressões de comportamento. Um importante mecanismo de controle nas linguagens funcionais é a aplicação recursiva de funções, o que, obviamente, facilitará a simulação das

especificações LOTOS, onde a recursão também é uma característica marcante. Além de alta ordem e recursão, há que se destacar também o polimorfismo e a composição de funções como outros importantes fatores a considerar na construção dos protótipos-simuladores de especificações LOTOS.

Em geral, as linguagens de programação funcionais possuem características como as que falamos acima, e se diferenciam por conceitos como módulos, tipos abstratos, tipos algébricos e, até mesmo, processos, entre outros.

4. SIMULAÇÃO

Neste item, apresentaremos nossas idéias e procedimentos usados para a confecção de protótipos que simulem especificações descritas em LOTOS. Para ilustrar e facilitar a compreensão mostraremos exemplos, como grau de dificuldade crescente, da passagem de especificações LOTOS para especificações executáveis em linguagens funcionais. (Usaremos uma "notação funcional" sem especificar qual linguagem estará sendo empregada.)

Em LOTOS, um processo é uma entidade capaz de executar *ações internas* (não observáveis) e de interagir com outros processos, que formam o seu *ambiente*. As interações se constituem de unidades de sincronização chamadas eventos ou ações. Eventos são *atômicos*, no sentido de que ocorrem instantaneamente. Quando dizemos que um processo executa uma *ação observável*, estamos dizendo que ocorre uma *interação* entre o processo e, ao menos, o observador. A definição de um processo especifica seu *comportamento* através da definição de *seqüências de ações observáveis*.

Estudando o texto do parágrafo acima, chegamos às seguintes definições:

- a entidade de execução em linguagens funcionais é a

função. Podendo assim, simular um processo LOTOS.

- as interações, que são formadas por unidades (eventos) elementares e atômicas de sincronização, formam o ambiente, que, desta forma, será representado por uma lista de eventos.
- os eventos, por sua vez, serão elementos de lista, representando execução instantânea.
- se a definição de um processo especifica que seu comportamento será dado por uma sequência de ações observáveis, então o resultado da execução de um processo (função) será uma lista de "ações (eventos) observáveis".

Das observações acima, já podemos concluir que o operador hide, por exemplo, que esconde (internaliza) um determinado conjunto de portas (eventos), tornando-as *não-observáveis*, pode ser simulado determinando-se que os elementos (portas) de uma dada lista não apareçam na lista que representará a execução do sistema, pois ela será a lista das ações *observáveis*. Da mesma forma, o operador de sequencialidade ";" é facilmente simulado usando-se a regra de formação de sequência (lista) das linguagens funcionais, ou seja, "*cabeça seguida de cauda*", que é representada por "(a:x)", e seria traduzida para "*evento seguido de expressão*".

Vejamos, então, um primeiro exemplo simples de uma simulação de um processo LOTOS, que usa apenas sequencialidade e recursão.

```
process Exemplo1 [a,b] : noexit :=
    a; b; Exemplo1 [a,b]
endproc
```

Como o nosso interesse é sempre observar o resultado final de uma simulação, esta será feita considerando-se um "ambiente fictício" que determinará o final da execução quando estiver esvaziado (lista vazia []).

```
exemplo1 [a,b] =
    a:b:exemplo1 [a,b]
```

Simulação sem parada

```
exemplo1 [a,b]      [] = []
exemplo1 [a,b] (c:amb) =
    a:b:exemplo1 [a,b] amb
```

Simulação com parada

Observe que na Simulação com parada, o ambiente é recebido como "(c:amb)" e na chamada recursiva ele é diminuído, passando a ser apenas "amb". Após sucessivas chamadas, o ambiente será reduzido até se tornar vazio ([]), encerrando a execução do processo (função).

De agora em diante chamaremos o ambiente fictício de **Ambiente de Controle**, que servirá, principalmente, para *ativar* e *desativar* a execução de processos.

Vamos, então, mostrar um exemplo completo de especificação, contendo os principais operadores de LOTOS. Os detalhes de nossas idéias a respeito da transformação dos operadores em construções funcionais estão em [FCM89], bem como, o modo como tratamos *não-determinismo*, que é simulado através de números aleatórios (não usado no exemplo). Entretanto, pelo fato da passagem da especificação para o protótipo ser quase direta, estamos certos de que sua compreensão será facilmente assimilada.

O exemplo trata de uma especificação de um sistema de alarme de carro [MoC89], que controla o funcionamento do automóvel através de um sinalizador de alarme e do mecanismo de abrir e fechar a porta do motorista. Seu funcionamento determina que o carro só poderá ser ligado (após o abrir e fechar da porta) se o alarme for sinalizado uma vez, e será mantido funcionando se houver uma segunda sinalização. Caso contrário, o carro irá parar de funcionar. Esta mesma situação ocorrerá se a porta for aberta e fechada com o carro funcionando, ou seja, se o alarme não for sinalizado após este procedimento, o carro irá parar. No entanto, se o carro for desligado e não houver abre/fecha da porta, ele poderá ser

ligado normalmente. Desta forma, são destacados os processos CONTROLE, responsável pelo funcionamento do alarme propriamente dito, e PAINEL, responsável pelo controle do sinalizador do alarme, do mecanismo de abre/fecha a porta e do mecanismo de liga/desliga a chave de ignição. Portanto, o processo PAINEL recebe do ambiente sinais do sinalizador de alarme (evento s), de liga ou desliga a chave de ignição (evento c) e abre/fecha a porta (evento p), passando-os ao processo CONTROLE. Logo, os dois processos se comunicam e podem ser colocados em paralelo, sincronizando a sinalização do alarme através do evento s e, os sinais de liga ou desliga a chave de ignição e abre/fecha a porta do carro, através de um evento g, que pode ser escondido do ambiente. Já podemos, então, começar a mostrar a especificação do comportamento do sistema, bem como do processo PAINEL, lembrando que não iremos considerar, no momento, as definições de tipos de dados envolvidos no problema (ACT ONE).

```
specification AlarmeCarro[s,c,p]  :  noexit  :=
  type ...
  end of def
behaviour
  hide  g  in
      PAINEL[s,c,p,g]  |[s,g]|  CONTROLE[s,g]
  where
  process PAINEL[s,c,p,g]  :  noexit  :=
          p!a-f ; g!port-af ; PAINEL[s,c,p,g]
      []  c!liga ; g!ch-liga ; PAINEL[s,c,p,g]
      []  c!desliga ; g!ch-desl ; PAINEL[s,c,p,g]
      []  s ; PAINEL[s,c,p,g]
  endproc
  ...
endspec
```

A definição funcional correspondente envolve o nosso ambiente de controle, que conterá a sequência de eventos escolhidos especificamente para a simulação.

```
alarmecarro [s,c,p] amb = hide [g] (par [s,g]
                                        (painel [s,c,p,g] amb)
                                        (controle [s,g] amb) )
```

O operador (função) hide retira do ambiente-resultado, gerado por par, todos os elementos(eventos) que estão na lista a esconder[1].

```
hide evesc       [] = []
hide evesc (e:ambr) = hide evesc ambr, pertce e evesc
                    = e : hide evesc ambr
```

A função par é um *árbitro* que define como será a intercalação ou comunicação dos eventos dos processos em paralelo. Por razão de espaço, a comunicação não será discutida aqui. (Estaria inserida na última equação de par.)

```
par ES B1          []  = B1
par ES []          B2  = B2
par ES (b1:B1)     B2  = b1 : par ES B2 B1, not (pertce b1 ES)
par ES (b1:B1) (b2:B2) = b2 : par ES (b1:B1) B2, pertce b1 ES
                                          and not (pertce b2 ES)
par ES (b:B1)  (b:B2)  = b  : par ES B1 B2, pertce b ES
```

A seguir definiremos a função painel, destacando que os eventos de comunicação são representados por listas cujo primeiro elemento (hd) é o nome da porta, o segundo (snd) é o "operador" (! ou ?) e o terceiro (trd) é o valor ou variável de interação, havendo ainda, um quarto elemento (não usado por não termos abordado *comunicação*), representado por uma função booleana indicando se a condição de sincronização satisfaz ou não.

```
painel [s,c,p,g]          []=[]    condição de parada (amb. fict.)
painel [s,c,p,g] (e:amb)=e:[g,!,port-af]:painel [s,c,p,g] amb,
                           hd e=p & snd e=! & trd e=port-af
                         =e:[g,!,ch-liga]:painel [s,c,p,g] amb,
                           hd e=c & snd e=! & trd e=ch-liga
                         =e:[g,!,ch-desl]:painel [s,c,p,g] amb,
                           hd e=c & snd e=! & trd e=ch-desl
                         =e:painel [s,c,p,g] amb, hd e=s
```

Vejamos mais uma parte da especificação relativa ao controle do alarme.

---

[1] pertce (pertence) corresponde ao operador matemático ∈.

```
process CONTROLE[s,g] : noexit :=
        ALARME[s,g] [] PARTIDA[s,g] [] SINAL[s,g]
where
  process SINAL[s,g] : noexit :=
          s ; CONTROLE[s,g]
  endproc
  process PARTIDA[s,g] : noexit :=
          g!ch-liga ; FUNC-OK[s,g]
  endproc
  process ALARME[s,g] : exit :=
          g!port-af ; FUNC-ALARME[s,g]
  where
     process FUNC-ALARME[s,g] : exit :=
             SINALIZA[s] >> FUNC-OK[s,g]
     where
        process SINALIZA[s] : exit :=
                s ; s ; exit
        endproc
     endproc
  endproc
...
```

A observação que temos a fazer é quanto ao operador de habilitação (>>) entre os processos SINALIZA e FUNC-OK (em FUNC-ALARME). Como SINALIZA apenas gera dois sinais de alarme e termina sua execução com sucesso (exit), sua simulação seria

```
sinaliza [s] = [s] : [s]
```

Esta lista deverá ser seguida pela execução de funcok. Optamos por fazer a concatenação (++) dos resultados (listas) de sinaliza e funcok para representar >> (esta é uma decisão não-generalizada), fazendo com que FUNC-ALARME seja definida funcionalmente como

```
funcalarme [s,g] amb = sinaliza [s] ++ funcok [s,g] amb.
```

Finalmente, vamos completar nossa especificação, definindo o processo FUNC-OK.

```
process FUNC-OK[s,g] : noexit :=
        g!ch-desl ; (g!port-af ; CONTROLE[s,g]
                     []
                     CONTROLE[s,g])
```

```
        []  g!port-af ; s ; FUNC-OK[s,g]
    endproc
...
```

Por ter um operador de escolha ([]) dentro de outro, preferimos definir esta situação através de uma função (esc) para representar o operador mais interno.

```
funcok [s,g] (e:amb) = e:esc [s,g] amb, hd e=g & snd e=!
                                        & trd e=ch-desl
                     = e:[s]:funcok [s,g] amb,hd e=g & snd e=!
                                        & trd e=port-af

esc [s,g] (e:amb) = e:controle [s,g] amb, hd e=g & snd e=!
                                        & trd e=port-af
                  = controle [s,g] amb
```

A definição completa do protótipo funcional correspondente à especificação LOTOS é dada a seguir. Observamos que hide e par não mais serão mostradas.

```
alarmecarro [s,c,p] amb = hide [g] (par [s,g]
                                    (painel [s,c,p,g] amb)
                                    (controle [s,g] amb))

painel [s,c,p,g]       []=[]
painel [s,c,p,g] (e:amb)=e:[g,!,port-af]:painel [s,c,p,g] amb,
                          hd e=p & snd e=! & trd e=port-af
                         =e:[g,!,ch-liga]:painel [s,c,p,g] amb,
                          hd e=c & snd e=! & trd e=ch-liga
                         =e:[g,!,ch-desl]:painel [s,c,p,g] amb,
                          hd e=c & snd e=! & trd e=ch-desl
                         =e:painel [s,c,p,g] amb, hd e=s

controle [s,g]       [] = []
controle [s,g] (e:amb) = alarme [s,g] (e:amb), hd e=g &
                                        snd e=! &
                                        trd e=port-af
                       = partida [s,g] (e:amb), hd e=g &
                                        snd e=! &
                                        trd e=ch-liga
```

```
                    = sinal [s,g] (e:amb),    hd e=s
sinal [s,g] (e:amb) = s : controle [s,g] amb
partida [s,g] (e:amb) = e : funcok [s,g] amb
alarme [s,g] (e:amb) = e : funcalarme [s,g] amb
funcalarme [s,g] amb = sinaliza [s] ++ funcok [s,g] amb
sinaliza [s] = [s] : [s]
funcok [s,g] (e:amb) = e:esc [s,g] amb, hd e=g & snd e=!
                                          & trd e=ch-desl
                     = e:[s]:funcok [s,g] amb,hd e=g & snd e=!
                                          & trd e=port-af
esc [s,g] (e:amb) = e:controle [s,g] amb, hd e=g & snd e=!
                                        & trd e=port-af
                  = controle [s,g] amb
```

5. CONCLUSÃO

Neste trabalho, falamos sobre a possibilidade de implementar os construtores de LOTOS usando linguagens funcionais, bem como gerar protótipos que simulem suas especificações. Os conceitos presentes na maioria das linguagens funcionais que nos fazem acreditar na viabilidade da simulação de LOTOS são, principalmente, listas, polimorfismo, alta ordem, composição e recursão. Assim, tendo-se os operadores de LOTOS implementados, a construção dos protótipos-simuladores pode ser feita de forma quase direta, como no exemplo mostrado.

REFERÊNCIAS

[BoB87] Bolognesi, T. and Ed Brinksma: "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, Vol 14, pp. 25-59, 1987.

[EhM85] Ehrig, H. and B. Mahr: "Fundamentals of algebraic specification", EATCS 6, Springer-Verlag, Berlin,

1985.

[FCM89] Ferraz, C., P. Cunha e S. Meira: "Ambientes de Especificação: Uma Comparação entre o Método Algébrico e o Método Funcional", Relatório Técnico, DI/UFPE, 1989.

[Gil87] Gilbert, D.: "Executable LOTOS: Using PARLOG to implement an FDT", 7th International Symposium on Protocol Specification, Testing, and Verification, Zurich, 1987.

[GHL88] Guillemot, R., M. Haj-Hussein and L. Logrippo: "Executing Large LOTOS Specifications", 8th Intl. Symp. on Protocol Specif., Testing, and Verification, Atlantic City, 1988.

[ISO88] ISO: "Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour", 8807, May 1988.

[Kar88] Karjoth, G.: "Implementing Process Algebra Specifications by State Machines", 8th Intl. Symp. on Protocol Specif., Testing, and Verification, Atlantic City, 1988.

[Mil80] Milner, R.: "A Calculus of Communicating Systems", LNCS 92, Springer-Verlag, Berlin, 1980.

[MoC89] Moura, T. e P. Cunha: "Desenvolvimento Estruturado de Especificações LOTOS", Relatório Técnico, DI/UFPE, 1989.