Modelagem e Verificação Formal de Sistemas Concorrentes: Um Tutorial Informal

Marcelo de Moura Amorim[1]

[1]Instituto de Matemática e Estatística – Universidade de São Paulo (USP)
Rua do Matão, 1010 - CEP 05508-090 - São Paulo - SP

mamorim@ime.usp.br

Resumo. *Este trabalho descreve um tutorial sobre modelagem e verificação formal de sistemas concorrentes utilizando como formalismo a álgebra de processos CSP e a ferramenta ProB para simulação e model checking. O objetivo é apresentar exemplos de específicações de modelos como introdução ao assunto e analisar o problema de deadlock dos sapos, além de propor uma variação ao problema e verificar se o deadlock ainda continua ocorrendo.*

1. Indrodução

O desenvolvimento de sistemas computacionais envolve o uso de modelos abstratos e precisos que permitem a uma equipe de engenheiros de software especificar, projetar, implementar e manter sistemas de software de modo a garantir os requisitos de qualidade pré-estabelecidos. Além destes aspectos técnicos, a engenharia de software envolve mecanismos para se planejar e gerenciar os processos de desenvolvimento de software, bem como as interações existentes entre as diversas pessoas que compõem uma equipe de desenvolvimento de software.

Este trabalho tem o objetivo de apresentar um tutorial sobre modelagem e verificação formal de sistemas concorrentes de uma forma mais simples e por meio do uso de exemplos. A partir de um exemplo de modelo simples que visa descrever o comportamento de um canibal chamado *aghori-par* e estendendo seu comportamento de modo a possibilitar a comunicação e execução concorrente com outro processo que descreve o comportamento de uma Lagoa, pretendemos formalizar o problema de deadlock dos sapos a ser apresentado na Seção 4. Além disso, será apresentada uma variação do problema a fim de verificar se a situação de *deadlock* ainda é mantida.

A Seção 2 apresenta um breve resumo da álgebra de processos CSP e sua sintaxe. A ferramenta Prob e notação CSPM utilizada neste texto será apresentada na Seção 3. Alguns exemplos utilizados para apresentar a especificação de sistemas concorrentes são apresentados na Seção 4. Por fim, concluímos esse trabalho com algumas considerações sobre os exemplos apresentados.

2. CSP

CSP (*Communicating Sequential Processes*), ou comunicação de processos seqüenciais, foi introduzido por CARHoare em 1978 [Hoare 1978]. Até o momento da publicação de seu livro, em 1985, [Hoare 1985] a evolução da linguagem foi influenciada de forma significativa. Desde então, no entanto, manteve-se praticamente estável,

sendo a única mudança significante passar a exigir que cada processo tenha um alfabeto de possíveis eventos definido, ou seja, são usados como uma parte integrante do operador de composição paralela. Segundo [Martin and Jassim 1997], o CSP fornece um excelente meio de descrever e raciocinar sobre os padrões de comunicação complexas, pelas seguintes razões:

- Ele incorpora os princípios fundamentais da comunicação de uma forma simples e elegante
- É semanticamente definido em termos de um modelo matemático rigoroso que pode ser utilizado para deduzir as propriedades do sistema formalmente
- Tem expressividade para permitir a verificação de *deadlock* e *livelock*
- Existem ferramentas automatizadas robustas para verificação formal em CSP [Roscoe 1995] [Malcolm et al. 1996]
- Algumas linguagens de programação, como occam, Ada e algumas variações da linguagem C para programação paralela são derivadas diretamente do modelo CSP e bibliotecas usando estilo CSP estão disponíveis para outras linguages, como o Java, etc.

O processo é a abstração utilizada para especificar comportamento. São construídos por meio de eventos, operadores e outros processos. Um sistema pode ser modelado por meio de um ou mais processos independentes que podem ser combinados para formar novos processos mais complexos: cada processo pode ser utilizado como fração do comportamento.

Sintaxe do Processo	Descrição
Stop	(quebra ou *deadlock*)
Skip	(término com sucesso)
$a \rightarrow P$	(prefixo)
$P(s)$	(recursão)
$P(s) \Box P(s)$	(escolha externa)
$P(s) \sqcap P(s)$	(escolha interna)
$if\,(g)\,then\,P(s)\,else\,P(s)$	(escolha condicional)
$P(s) \setminus C$	(internalização)
$P(s)\,;\,P(s)$	(composição seqüencial)
$P(s) \triangle P(s)$	(interrupção)
$P(s) \|[C]\| P(s)$	(paralelismo)

Figura 1. Processos primitivos e operadores algébricos de CSP

Um evento é um abstração para a ocorrência de um fato real, é o objeto mais elementar de CSP. Um evento pode pertencer a uma classe de valores denominada canal. Um canal representa uma coleção de eventos com características comuns. Eventos fazem parte de um alfabeto, denotado pela letra Σ, que contém todas as possíveis comunicações para os processos dentro do universo considerado. A ocorrência de um evento em um

processo caracteriza uma comunicação deste processo com pelo menos um participante. Geralmente o participante é um outro processo, caso contrário será o próprio ambiente em que o processo está inserido. A comunicação entre processos é atômica e se dá através de passagem de mensagens simultâneas. Como o modelo de comunicação é síncrono, todos os processos participantes devem estar simultaneamente prontos para executar a comunicação. A composição de eventos para formar um processo e o relacionamento entre diferentes processos é descrita através dos operadores algébricos de CSP. A sintaxe de CSP define a forma como eventos e processos podem ser combinados através de operadores para formar novos processos. A Figura 1 apresenta a sintaxe do CSP.

3. CSPM e Prob

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (Failures and Divergences Refinament [FDR 2005]. O ProB [Systems 2003] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSPM [Systems 2003], que é um acrônimo para Machine Readable CSP. O Prob é empregada dentro deste trabalho para verificar a validade dos refinamentos propostos e permite simular o comportamento passo a passo das especificações CSP que serão exemplificadas.

A Figura 2 apresenta a sintaxe do CSP na notação CSPM e a relação dos operadores de processo listados na Seção anterior.

Sintaxe do Processo	Descrição
STOP	(quebra ou *deadlock*)
SKIP	(término com sucesso)
a -> P	(prefixo)
$P(s)$	(recursão)
$P(s)$ [] $P(s)$	(escolha externa)
$P(s)$ |~| $P(s)$	(escolha interna)
if (g) then $P(s)$ else $P(s)$	(escolha condicional)
$P(s) \setminus C$	(internalização)
$P(s)$; $P(s)$	(composição seqüencial)
$P(s)$ /\ $P(s)$	(interrupção)
$P(s)$ [|C|] $P(s)$	(paralelismo)

Figura 2. Sintaxe do CSP na notação CSPM e a relação dos operadores de processo

4. Exemplos de especificações em CSP e CSPM

A fim de apresentar alguns exemplos de modelagem e verificação formal de sistemas concorrentes foram gerados alguns exemplos que pudessem ser de fácil entendimento pelo leitor. Ao final, a idéia é apresentar uma especificação para o problema de deadlock dos

sapos e propor uma variação ao problema executando concorrentemente com o processo *aghori-par*. Para isso, no primeiro exemplo, assumimos de forma fictícia, a existencia de uma tribo chamada AGHORIS-PAR. Em seguida, apresentamos um exemplo de modelo de uma lagoa. E por fim, apresentamos o modelo do problema dos sapos e sua variação estendida.

4.1. AGHORIS-PAR

AGHORIS-PAR são membros de uma seita hindu que adoram Shiva e computação paralela. Possuem um hábito estranho:

1. Adoram ficar pensando
2. Quando encontra outro Aghori-par pode cantar
3. Quando cantam ficam com fome
4. Só voltam ao normal quando comem sapos

```
-- aghori

channel pensar, comerSapo, cantar, fome

AGHORI = pensar -> AGHORI [] cantar -> fome -> comerSapo -> AGHORI
AGHORI_GRUPO = AGHORI ||| AGHORI
AGNORI_GRUPO2 =  AGHORI [|{|cantar|}|] AGHORI [|{|cantar|}|] AGHORI
```

Figura 3. Modelo na notação CSPM de um processo representando um aghori-par

Conforme a Figura 3, são criados os canais $pensar, comerSapo, cantar, fome$ pelos quais o processo poderá se comunicar com outros processos. Um AGHORI pode então pensar e voltar a se comportar como AGHORI ou cantar, sentir fome, comer um sapo e voltar a se comportar como AGHORI. $AGHORI|||AGHORI$ representa dois AGHORI sendo executados concorrentes sem comunicação entre eles e $AGHORI[||cantar||]AGHORI$ dois processos AGHORI sendo executados de maneira a possibilitar a comunicação entre os mesmos.

4.2. MODELO LAGOA CSP

O modelo de AGHORI apresentado na Seção anterior ficaria ainda mais interessante, se pudessemos executá-lo em paralelo com outro processo que produzisse sapos. Desta forma, um AGHORI poderia cantar, sentir fome, comer um sapo e voltar a se comportar como AGHORI.

Desta forma, podemos assumir:

"SEMPRE QUE CHOVE UM SAPO SERÁ GERADO EM UMA LAGOA".

A Figura 4 apresenta o modelo CSPM do comportamento da Lagoa. Os canais $chuva, sol$ representa as ações de chover e fazer sol respectivamente. Uma lagoa se comporta como: $LAGOA = sol \rightarrow LAGOA\ [\ ]\ chuva \rightarrow NASCE\_SAPO$, isto é, pode executar a ação de sol e voltar a se comportar como LAGOA ou ação de chuva e se comportar como NASCE_SAPO. O comportamento de NASCE_SAPO é descrito pela possibilidade de executar comerSapo e entrar em deadlock $STOP$ concorrentemente voltando a se comportar como LAGOA.

```
-- aghori
-- lagoa

channel pensar, comerSapo, cantar, fome
channel chuva, sol

AGHORI = pensar -> AGHORI [] cantar -> fome -> comerSapo -> AGHORI
AGHORI_GRUPO = AGHORI ||| AGHORI
AGNORI_GRUPO2 =  AGHORI [|{|cantar|}|] AGHORI
AGNORI_LAGOA =  ( AGHORI [|{|cantar|}|] AGHORI ) [|{|comerSapo|}|] LAGOA
LAGOA = sol -> LAGOA [] chuva -> NASCE_SAPO
NASCE_SAPO = comerSapo -> STOP ||| LAGOA
```

Figura 4. Modelo na notação CSPM de um processo representando uma Lagoa

4.3. O PROBLEMA DE DEADLOCK DOS SAPOS

O problema dos sapos pode ser modelado com quatro sapos SE_1, SE_2, SD_1, SD_2 organizados em uma sequencia de cinco pedras $P_1, P_2, ..., P_5$, sendo que dois deles estão nas duas primeiras pedras e podem apenas andar (pular) para direita, e os outros dois estão posicionados nas últimas pedras e apenas podem saltar para a direita. A Figura 5 apresenta a configuração inicial dos sapos. Os sapos podem apenas saltar para a pedra vazia logo ao seu lado, ou, saltar um sapo a sua frente caso exista uma pedra vazia a frente do sapo.

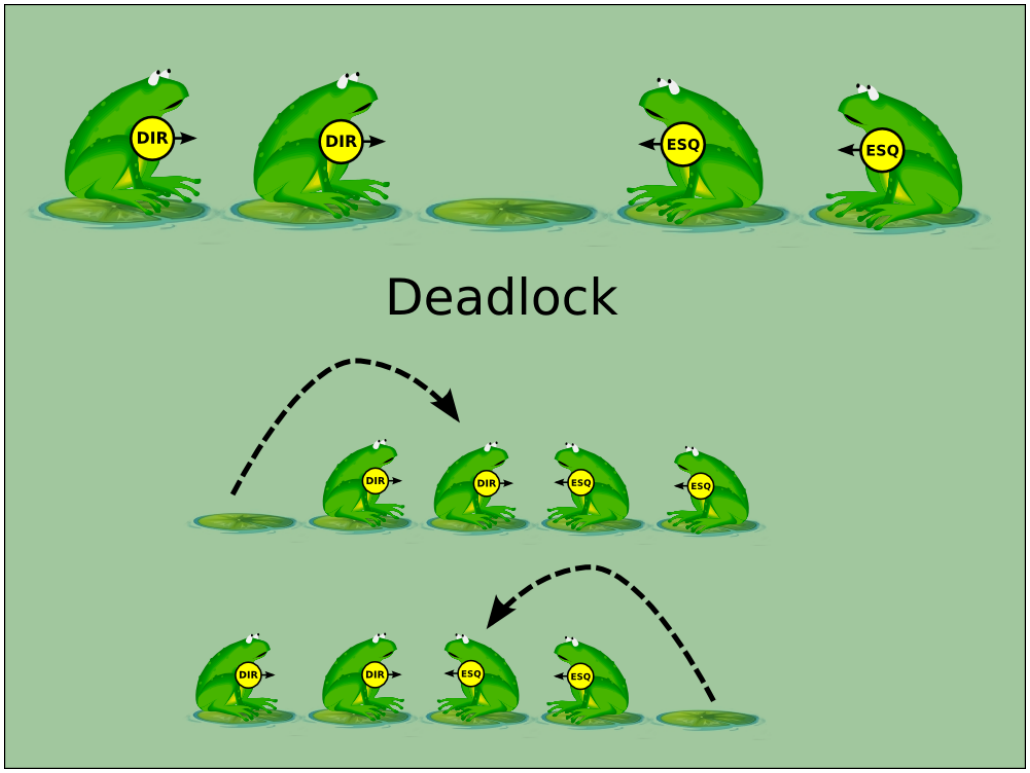

Figura 5. Modelo na notação CSPM de um processo representando uma Lagoa

A situação de deadlock acontece quando no início do processo o sapos que estão na extremidade saltam sobre o sapo a sua frente, isto é, o sapo SE_1 salta o sapo SE_2 e

fica posicionado na pedra P_3 ou de forma análoga, o sapo SD_2 salta o sapo SD_1 e fica posicionado na pedra P_3. Entretanto, existem casos que o sistema não entra em *deadlock* conforme a execução da sequência de passos:

$$\overrightarrow{SE_2\ P_3} \rightarrow \overrightarrow{SD_4\ P_2} \rightarrow \overrightarrow{SD_5\ P_4} \rightarrow \overrightarrow{SD_2\ P_2} \rightarrow \overrightarrow{SE_1\ P_3} \rightarrow \overrightarrow{SD_1\ P_1} \rightarrow \overrightarrow{SD_2\ P_2} \rightarrow \overrightarrow{SE_2\ P_4} \rightarrow SKIP$$

A Figura 6 apresenta a especificação em CSPM do sistema.

```
-- sapos VERSÃO 1

ITENS = {1..5}
datatype SAPO = sapo
channel pedraLivre: ITENS.SAPO.ITENS
channel novaPedra: ITENS
channel feliz

inSet(x,s) = inter(s,{x}) == {x}
remSet(x,s) = diff(s,{x})

Pedras(s) = ([] i:s @ pedraLivre.i.sapo?x -> Pedras(union(remSet(i,s),{x}))) []
feliz -> SaposFelizes []
novaPedra?pi -> Pedras(union(s,{pi}))

Sapos(saposD,saposE) = ([] i:saposD @ SaposDir(i,saposD,saposE)) []
       ([] i:saposE @ SaposEsq(i,saposD,saposE)) []
if(saposD == {} and saposE == {}) then
   feliz -> SaposFelizes
else
   STOP

SaposFelizes = feliz -> SaposFelizes

SaposDir(i,sD,sE) = if(i == 5) then novaPedra!i -> Sapos(remSet(i,sD),sE) else STOP []
    pedraLivre.i+1.sapo!i -> Sapos(union(remSet(i,sD),{i+1}),sE) []
    if(inSet(i+1,union(sD,sE))) then
pedraLivre.i+2.sapo!i -> Sapos(union(remSet(i,sD),{i+2}),sE)
    else
STOP

SaposEsq(i,sD,sE) = if(i == 1) then novaPedra!i -> Sapos(sD,remSet(i,sE)) else STOP []
    pedraLivre.i-1.sapo!i -> Sapos(sD,union(remSet(i,sE),{i-1})) []
    if(inSet(i-1,union(sD,sE))) then
pedraLivre.i-2.sapo!i -> Sapos(sD,union(remSet(i,sE),{i-2}))
    else
STOP

MAIN = Pedras({3}) [|{|novaPedra,feliz,pedraLivre|}|] ( Sapos({1,2},{4,5}) )
```

Figura 6. Modelo na notação CSPM do sistema representando o problema dos sapos

O modelo apresentado pode então ser veirificado com relação à deadlock utili-

zando a ferramenta ProB com a diretiva $assert\ MAIN\ [T =\ TRACE$, conforme a Figura 7. Podemos também verificar que a execução da sequência de passos descrita por TRACES é possível de ser executada.

```
TRACE = pedraLivre.3.sapo!2 -> pedraLivre.2.sapo!4 -> pedraLivre.4.sapo!5 -> pedraLivr
assert MAIN [T= TRACE
```

Figura 7. Modelo na notação CSPM de um processo representando uma Lagoa

4.4. VARIAÇÃO DO PROBLEMA COM AGHORI-PAR

Com o objetivo de estudar uma possível variação do problema afim de tentar eliminar a situação de deadlock do sistema, propomos uma estensão do mesmo. Assim, o problema dos sapos pode ser modelado de forma que execute concorrente com um AGHORI conforme a Figura 8.

Figura 8. Variação do Modelo de deadlock dos sapos com processo aghori-par

Neste caso, teríamos a especificação do sistema conforme a Figura 8.

```
MAIN = Pedras({3}) [|{|novaPedra,feliz,pedraLivre|}|] ( Sapos({1,2},{4,5}) [|{|comerSa
MAIN2 = Pedras({3}) [|{|novaPedra,feliz,pedraLivre|}|] ( Sapos({1,2},{4,5}) [|{|comerS

assert MAIN [T= TRACE

channel pensar, comerSapo, cantar, fome

AGHORI = pensar -> AGHORI [] cantar -> fome -> comerSapo -> AGHORI
AGHORI2 = pensar -> AGHORI [] cantar -> fome -> comerSapo -> STOP
```

Figura 9. Modelo na notação CSPM do sistema concorrente com aghori-par

Utilizando a ferramenta ProB, podemos verificar que a variação continua ocorrendo *deadlock* quando especificamos o comportamento do AGHORI para que possa comer apenas um sapo:

$$AGHORI2 = pensar \rightarrow AGHORI\ [\,]\ cantar \rightarrow fome \rightarrow comerSapo \rightarrow STOP$$

Entretanto, quando especificamos um AGHORI de forma que ele volte a se comportar como AGHORI após comer um sapo o *deadlock* deixa de ocorrer. Neste caso a especificação volta a ser como:

$$AGHORI = pensar \rightarrow AGHORI \; [\,] \; cantar \rightarrow fome \rightarrow comerSapo \rightarrow AGHORI$$

Utilizando a ferramenta ProB, podemos verificar esta situação com as diretivas apresentadas na Figura 10.

```
-- Se comer um sapo tudo fica feliz
{-# assert_ltl "F(e(comerSapo) => e(feliz))" #-}

-- Se NÃO comer um sapo tudo fica feliz
{-# assert_ltl "F(not e(comerSapo) => e(feliz))" #-}
```

Figura 10. Diretivas lógica LTL para asserção de deadlock free

5. Conclusão e Considerações

O trabalho apresentado buscou descrever um tutorial com o objetivo de apresentar de forma simples como é realizada a modelagem e verificação formal de sistemas concorrentes usando uma álgebra de processo.

Partindo de exemplos mais simples e chegando à um problema um pouco mais complexo envolvendo a situação de *deadlock*, procurou-se desenvolver os conceitos práticos de como é possível utilizar métodos formais no auxílio da implementação de programas que estejam corretos a partir de um modelo préviamente provado (verificado automatizadamente).

Este tutorial foi apresentado como seminário durante as aulas da disciplina MAC-5742 COMPUTAÇÃO PARALELA E DISTRIBUÍDA ministrada pelo professor Dr. Alfredo Goldman no Instituto de Matemática e Estatística da Universidade de São Paulo durante o segundo semestre de 2015.

Referências

FDR (2005). Fdr failures-divergence refinement - fdr2 user manual.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Malcolm, J., Martin, R., and Martin, J. (1996). The design and construction of deadlock-free concurrent systems.

Martin, J. and Jassim, S. (1997). How to design deadlock-free networks using csp and verification tools - a tutorial introduction.

Roscoe, A. W. (1995). Modelling and verifying key-exchange protocols using csp and fdr. In *In 8th IEEE Computer Security Foundations Workshop*, pages 98–107. Press.

Systems, F. (2003). Probe user manual formal systems (europe) ltd.