ESPECIFICAÇÃO FORMAL DE SISTEMAS DISTRIBUIDOS USANDO Z E SISTEMAS DE TRANSIÇÃO

Carmen Huamalí Salazar
Silvio Romero de Lemos Meira

Departamento de Informática
Centro de Ciências Exatas e da Natureza
Universidade Federal de Pernambuco
50739, Recife PE, BRASIL

ABSTRACT

We show a Z and temporal logic based formal specification technique that allows for the expression, in a modular way, of the behavior and the structure of distributed systems.

RESUMO

Aqui mostramos uma técnica de especificação formal baseada em Z e lógica temporal que permite expressar de forma modular o comportamento e a estrutura de sistemas distribuidos.

1 INTRODUÇÃO

1.1 POR QUE LINGUAGENS FORMAIS?

Linguagens naturais são muito expressivas mas imprecisas. Com linguagens formais não podemos dizer muito, mas o que dizemos não é ambigüo.

Técnicas de especificação e desenvolvimento de programas que usam linguagens formais tem um papel importante nos projetos de software, pois:

- Permitem abstrair a especificação do contexto de implementação.
- Estabelecem uma notação precisa na qual tais projetos podem ser facilmente comunicados a outras pessoas.
- É possivel apresentar a especificação diretamente no manual do usuário de sistemas de software.
- Permitem construir especificações em diferentes niveis de abstração.

Nos sistemas já existentes, a especificação pode ser usada:

- Na re-implementação. A re-implementação de um sistema pode envolver a incorporação de novas técnicas, mudanças evolutivas, assim como a adição ou remoção de componentes ou mudanças operacionais. Como a especificação é independente da implementação, ela proverá um ponto

de partida para cada caso.

- Como uma base na alteração de um sistema. Quando mudanças ou adições ao sistema são feitas, novas especificações podem ser desenvolvidas com referência a especificações prévias. Este desenvolvimento nos dá uma visão dos efeitos das mudanças e de sua interação com partes existentes do sistema.
- Experimentos com especificações provêm um método mais rápido e barato de investigar possíveis mudanças que um sistema pode sofrer.

1.2 QUE É Z?

É amplamente reconhecido que, para construir um sistema de software de grande porte, é necessário dividir o sistema em componentes e construí-lo como uma configuração desses componentes. Em Z[3], uma técnica formal de especificação e desenvolvimento de programas, isto é refletido na existência de duas linguagens complementares: a linguagem matemática e a linguagem de esquemas. A linguagem matemática está baseada em cálculo de predicados e proposicional mas é "fortemente tipificada". A linguagem de esquemas suporta a apresentação sistemática de especificação de sistemas de larga escala ou família de especificações e incorpora definições da linguagem matemática.

1.3 POR QUE RACIOCÍNIO TEMPORAL?

Na especificação e raciocínio sobre programas seqüenciais tem sido usado por muitos anos a lógica formal ordinária de primeira ordem. Por que a introdução de concorrência requer o uso de raciocínio temporal?

Para descrever um programa seqüencial, precisa-se considerar os estados de dois tempos: antes e depois que o programa seja executado. O programa pode ser considerado como a função desde um estado inicial a um final, e assim pode ser considerado em termos de condições de entrada e saída. O raciocínio temporal simples antes/depois que é usado não precisa ser explícito.

Isto não é verdade para programas concorrentes. Por exemplo, considere as duas sentenças do programa abaixo, onde o <> denota operações atômicas. Uma operação atômica de um programa concorrente é indivisível com respeito a operações executadas concorrentemente.

```
S1: <x := x + 1>
S2: begin
      <x := x + y + 1>
      <x := x - y>
    end
```

As sentenças S1 e S2 produzem o mesmo mapeamento de estado inicial a final. Elas satisfazem exatamente as mesmas condições de entrada e saída, e são equivalentes quando usadas em um programa seqüencial. Contudo, considere o seguinte programa concorrente, onde o cobegin indica que suas duas cláusulas estão sendo executadas concorrentemente.

cobegin < y := y - 7 > ‖ S coend

Sustituir S por S1 produz um programa que incrementa x por um. No entanto, sustituindo S por S2 produz um programa que incrementa x por um ou oito. A última possibilidade ocorre se a sentença <y := y - 7> é executada entre as duas sentenças de S2. Por tanto, S1 e S2, que satisfazem iguais condições de entrada/saída, não são equivalentes quando aparecem como parte de um programa concorrente.

Este exemplo mostra que quando descrevemos uma propriedade concorrente, não podemos restringir nossa atenção ao que é verdade antes e depois de uma execução. Devemos considerar também que acontece durante sua execução.

1.4 QUE SE PODE ESPECIFICAR EM LÓGICA TEMPORAL?

Lógica temporal é uma linguagem formal para expressar propriedades temporais. Pnueli foi um dos primeiros a reconhecer a utilidade de lógica temporal para raciocinar sobre propriedades de programas concorrentes[5].

Lógica temporal nos permite expressar duas classes de propriedades de programas:

- Propriedades de segurança, que garantem que o programa não faz nada errado. Isto é, o programa nunca atinge um estado inaceitável.
- Propriedades de justiça[6], que asseguram que o programa eventualmente fará algo correto. Isto é, o programa eventualmente atingirá um estado desejavel.

Correção parcial (o programa não produz resposta errada), exclusão mútua (dois processos não estão em sua seção crítica ao mesmo tempo) e ausência de "deadlock" (o programa não atinge um estado de espera mútua infinita) são algumas propriedades de segurança. Terminação (o programa eventualmente termina), ausência de "starvation" (o processo eventualmente recebe serviço), atendimento eventual de toda solicitação de serviçoe e a entrada eventual de um processo na sua seção crítica são exemplos de propriedades de justiça.

Programas seqüenciais só apresentam correção parcial como propriedade de segurança e terminação é a unica propriedade de justiça que tem

recebido cuidadoso tratamento formal.

Lógica temporal foi desenvolvida para descrever a ordem ao invés do tempo, na qual os acontecimentos devem ocorrer.

1.5 A LINGUAGEM DE ESPECIFICAÇÃO

Aqui nos apresentamos um modelo baseado em transições para especificar formalmente de forma modular, seguindo a idéia de esquema em Z, o comportamento e a estrutura do sistema. As propriedades de segurança e justiça são especificadas usando asserções em lógica temporal. Mostramos alguns exemplos de especificações usando o modelo.

2. MODELO CONCEITUAL PARA SISTEMAS DISTRIBUIDOS

Um sistema pode ser descrito de dois diferentes pontos de vista[2]. Do ponto de vista do projetista, o sistema consiste de interação de processos locais com usuários e comunicação entre processos via o meio. Cada processo local pode ser descrito pelas operações respondendo a comandos e mensagens de outros processos. Este é ponto de vista da estrutura do sistema.

Do ponto de vista do usuário, um sistema distribuído é um servidor compartilhado, ou uma caixa preta com só uma interface visível. Neste caso, exceto por questões de performance, não há nenhuma diferença funcional entre um sistema distribuído e um centralizado. A única coisa interessante são as relações entre as mensagens ou eventos acontecendo na interface. Esta classe de descrição de interface de um sistema é chamada descrição comportamental.

3. DESCRIÇÃO DO COMPORTAMENTO

3.1 MODELO DE TRANSIÇÕES

Representamos a execução de um programa concorrente como uma seqüência de transições de estado da forma

$$S \xrightarrow{\alpha} S'$$

denotando que a ação atômica α leva o programa de estado S ao estado S'. Esta transição pode representar a execução de uma operação indivisível[1] em algum processo, na qual α denota a operação do programa sendo executado, S o estado antes da execução de α e S' o estado imediatamente após da execução. Contudo a natureza exata dos estados e ações não nos interessa. A seqüência de transições do sistema

---

[1] Asumimos que uma referencia de memoria é indivisível, se dois rocessos intentam referenciar à mesma celula de memoria no mesmo tempo, resultado é como se a referencia fuese feita serialmente.

compreende a intercalação das seqüências das ações atômicas dos processos componentes individuais. Assim concorrência é modelada pela intercalação de operações atômicas concorrentes.

Caracterizamos o programa pelo conjunto de todas as possíveis seqüências de execução. Este conjunto contém execuções iniciando de qualquer estado. Para um programa não determinístico, pode haver muitas seqüências com o mesmo estado inicial.

Um estado consiste de um possível "snapshot" tomado no meio de uma execução, contendo toda a informação necessária para resumir a correta execução desde esse ponto. Assim, o estado deve descrever o valor de cada variavel de dado, de canal, etc., do processo.

Então, a sequência

$$S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} S_2 \xrightarrow{\alpha_3} \ldots$$

descreve uma execução que começa no tempo zero em estado S_0, está no estado S_1 no tempo um, etc. O tempo zero é visto como presente, todo outro tempo como futuro.

3.2 ESPECIFICAÇÃO

Para especificar um programa, devemos especificar o conjunto de todas suas possíveis seqüências de execução. A especificação consiste de uma coleção de condições (asserções) das seqüências de execuções. Um programa satisfaz a especificação se todas as suas possíveis seqüências de execução satisfazem cada uma dessas condições.

Especificamos dois tipos de propriedades, as de segurança ("safety") descrevendo o que o sistema é permitido fazer e as de justiça ("liveness") descrevendo o que o sistema deve fazer.

A propriedade de segurança será dada pela descrição das possíveis transições e os eventos associados a estas. Para cada transição, definimos a relação entre o estado antes e depois da transição. A parte de justiça estabelece as condições sobre as quais a transição deve eventualmente ser ativada.

As propriedades são mantidas durante a execução do programa. Podemos descrever que significa para uma asserção em lógica temporal P ser verdade em "tempo i" na seqüência de execução

$$S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} S_2 \xrightarrow{\alpha_3} \ldots$$

quando o estado do programa é S_i, usando quatro acerções:

P : Verdade em tempo i se e só se P é verdade no estado Si

$\Box$ P : Verdade em tempo i se e só se P é verdade em todo tempo $j \geq i$

PDF compressor

$\Diamond P$: Verdade em tempo i se e só se P é verdade em algum tempo $j \geq i$

$\Box \Diamond P$: Verdade em tempo i se e só se P é verdade infinitamente muitas vezes para tempo $j \geq i$.

Pronunciamos $\Box$ como "sempre", $\Diamond$ como "eventualmente", e $\Box\Diamond$ "infinitamente amiúde". Por exemplo, a propriedade $\Box P \rightarrow \Diamond Q$ (se lê "P conduz a Q") afirma que em algum tempo i, se P é verdade para todo estado S_j com $j \geq i$ logo Q é verdade para algum estado S_j com $j \geq i$.

A tautologia

$$\sim \Box P \equiv \Diamond \sim P$$

é importante. Estabelece que P é nem sempre verdade se e só se ele é eventualmente falso.

Em seguida apresentamos alguns exemplos de especificações usando o modelo de transições.

Buffer Limitado

No problema de buffer limitado, distinguimos o buffer como sendo o servidor com os clientes produtor e consumidor.

Um buffer pode ser modelado como uma seqüência de carateres

buffer : seq char

e um buffer limitado pode ser definido pelo esquema (em Z[3])

```
— LBUFFER ——————————————————————————
 buffer   : seq char
 max_size : N
 ———————————————————————————————————
      0 ≤ # buffer ≤ max_size
 ———————————————————————————————————
```

Um esquema tem duas partes: as declarações (acima da linha central) na qual variáveis que são usados no esquema são declaradas, e o predicado (abaixo da linha central) contendo predicados dando propriedades e relacionando essas variáveis.

No esquema LBUFFER, declaramos buffer como uma seqüência de carateres e a variável inteira de nome *max_size* que indica a capacidade do buffer. A parte do predicado estabelece que o tamanho do buffer (# buffer) não deve passar de sua capacidade nem ser menor que 0 (0≤#buffer≤max_size).

O estado inicial do buffer é dado por uma seqüência vazia

LBUFFER-INITIAL ≙ [LBUFFER ! buffer = [ ]]

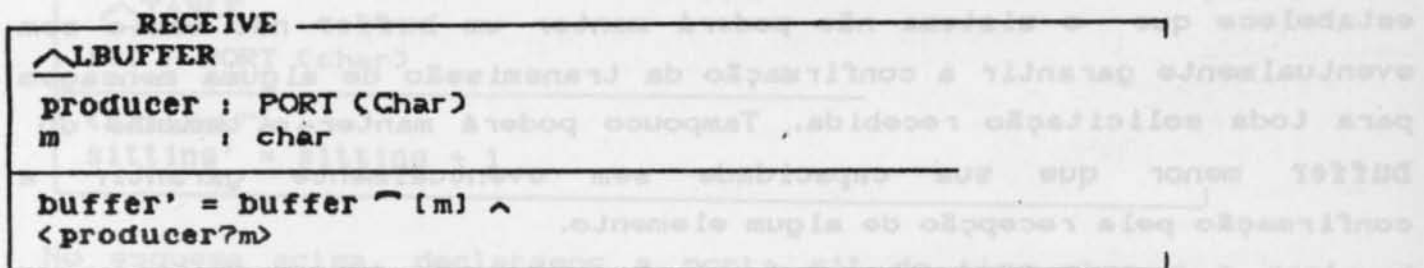
Na definição das transições de um buffer usamos o esquema:

⌢LBUFFER ≙ LBUFFER ⌢ LBUFFER'

(⌢para mudança) define o estado do buffer antes da operação LBUFFER,

com componente buffer (satisfazendo $0 \le \# buffer \le max_size$), e um estado depois LBUFFER' com componente buffer' (satisfazendo $0 \le \# buffer' \le max_size$).

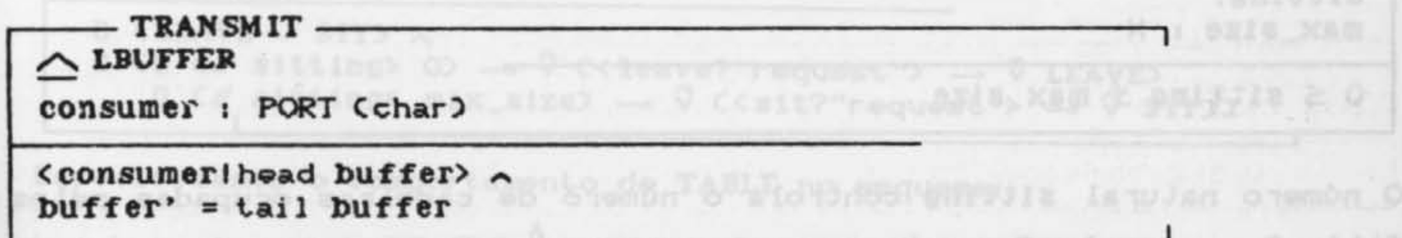
Definimos as possíveis mudanças de estado e os eventos associados:



```
—— RECEIVE ————————————————————————
ΔLBUFFER
producer : PORT (Char)
m        : char
———————————————————————————————————
buffer' = buffer ⁀ [m] ∧
<producer?m>
———————————————————————————————————
```

O esquema ΔLBUFFER é usado para descrever os estados do buffer antes e depois da transição. Quando um esquema é incluído na parte declarativa de outro, suas declarações passam a fazer parte do esquema em pauta e há uma conjunção dos predicados.

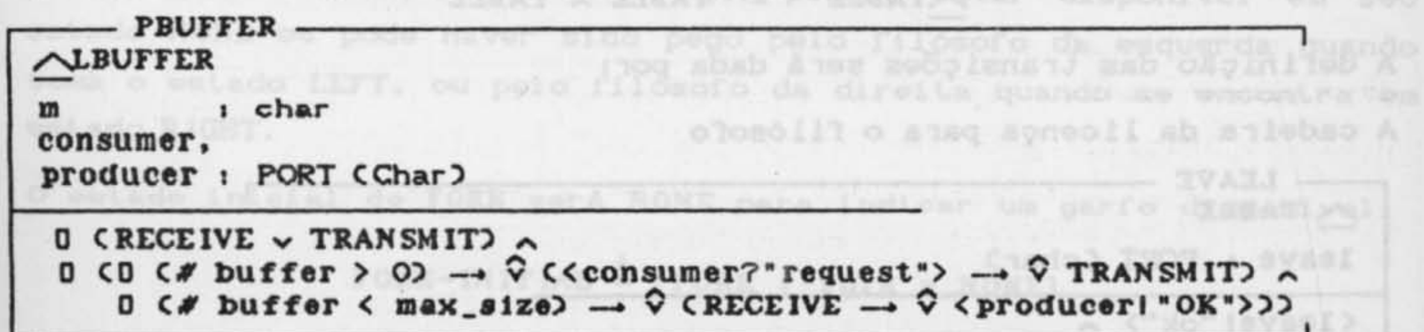
No esquema RECEIVE, declaramos producer como sendo uma porta de tipo char. O predicado do esquema estabelece que um valor m é lido da porta producer (<producer?m>), o qual é colocado no buffer (buffer' = buffer ⁀ [m]).



```
—— TRANSMIT ———————————————————————
Δ LBUFFER
consumer : PORT (char)
———————————————————————————————————
<consumer!head buffer> ∧
buffer' = tail buffer
———————————————————————————————————
```

Declaramos a porta consumer de tipo char. O predicado especifica que a ocorrência de um evento de saída transmitindo o primeiro elemento do buffer (<consumer!head buffer>) causa a remoção do elemento da cabeça do buffer (buffer' = tail buffer).

Especificamos as propriedades do buffer no esquema:



```
—— PBUFFER ————————————————————————
ΔLBUFFER
m        : char
consumer,
producer : PORT (Char)
———————————————————————————————————
□ (RECEIVE ∨ TRANSMIT) ∧
□ (□ (# buffer > 0) → ◇ (<consumer?"request"> → ◇ TRANSMIT) ∧
   □ (# buffer < max_size) → ◇ (RECEIVE → ◇ <producer!"OK">))
———————————————————————————————————
```

A propriedade de segurança da especificação, primeira linha do

predicado, descreve as transições que deve manter o sistema. A propriedade de segurança não implica que as transições realmente sejam realizadas. Para isto precisamos da propriedade de justiça.

A propriedade de justiça, que assegura o acesso ao buffer, é expressa pela fórmula temporal da segunda e terceira linhas do predicado e estabelece que o sistema não poderá manter um buffer não vazio sem eventualmente garantir a confirmação da transmissão de alguma mensagem para toda solicitação recebida. Tampouco poderá manter o tamanho do buffer menor que sua capacidade sem eventualmente garantir a confirmação pela recepção de algum elemento.

Podemos especificar o comportamento do buffer numa única fórmula temporal, num nível mais abstrato, como:

$$BOUND \triangleq LBUFFER\text{-}INITIAL \wedge PBUFFER$$

Os Filósofos Jantando

O segundo exemplo é o famoso problema dos filósofos jantando[2]. Aqui os servidores serão o garfo e a cadeira, enquanto que o filósofo será o cliente.

Um servidor de cadeiras será definido por:

```
── TABLE ─────────────────────────────
 sitting,
 max_size : N
──────────────────────────────────────
 0 ≤ sitting ≤ max_size
──────────────────────────────────────
```

O número natural sitting controla o número de cadeiras ocupadas pelos filósofos o qual não pode ser maior que max_size, número de cadeiras existentes na mesa.

O estado inicial da mesa é dado pelo número de filósofos sentados igual a zero.

$$TABLE_INITIAL \triangleq [TABLE \mid sitting = 0]$$

Definimos duas transições em TABLE, nas quais usamos o esquema:

$$\triangle TABLE \triangleq TABLE \wedge TABLE'$$

A definição das transições será dada por:

A cadeira da licença para o filósofo

```
── LEAVE ─────────────────────────────
 △ TABLE
 leave : PORT (char)
──────────────────────────────────────
 <leave! "ok"> ∧
 sitting' = sitting - 1
──────────────────────────────────────
```

Declaramos a porta leave, a parte do predicado especifica que há um evento de saída confirmando (<leave!"ok">) o qual faz diminuir o número de cadeiras ocupadas (sitting'=sitting - 1).

A cadeira da licença para que o filósofo se sente

```
── SIT ──────────────────────────────
 ΔTABLE
 sit : PORT (char)
─────────────────────────────────────
 <sit!"ok"> ∧
 sitting' = sitting + 1
─────────────────────────────────────
```

No esquema acima, declaramos a porta sit de tipo char e a parte do predicado estabelece que dado um evento de saída confirmando (<sit!"ok">) haverá um aumento do número de cadeiras ocupadas. ΔTABLE garante que $0 \leq$ sitting' $\leq$ max_size.

As propriedades de um servidor de cadeiras são análogas às do buffer limitado, assim especificamos o esquema:

```
── PTABLE ───────────────────────────
 ΔTABLE
 sit,
 leave : PORT (Char)
─────────────────────────────────────
 □ (LEAVE ∨ SIT) ∧
 □ (□ (# sitting> 0) → ◇ (<leave?"request"> → ◇ LEAVE)
    □ (# siting< max_size) → ◇ (<sit?"request"> → ◇ SIT))
─────────────────────────────────────
```

Especificamos o comportamento de TABLE no esquema:

$$DTABLE \triangleq TABLE\text{-}INITIAL \wedge PTABLE$$

Os garfos são especificados por seus tres estados

```
── FORK ─────────────────────────────
 fork : LEFT | RIGHT | NONE
─────────────────────────────────────
```

O esquema FORK modela um garfo que poderá estar disponível em seu estado NONE ou pode haver sido pego pelo filósofo da esquerda quando toma o estado LEFT, ou pelo filósofo da direita quando se encontra em estado RIGHT.

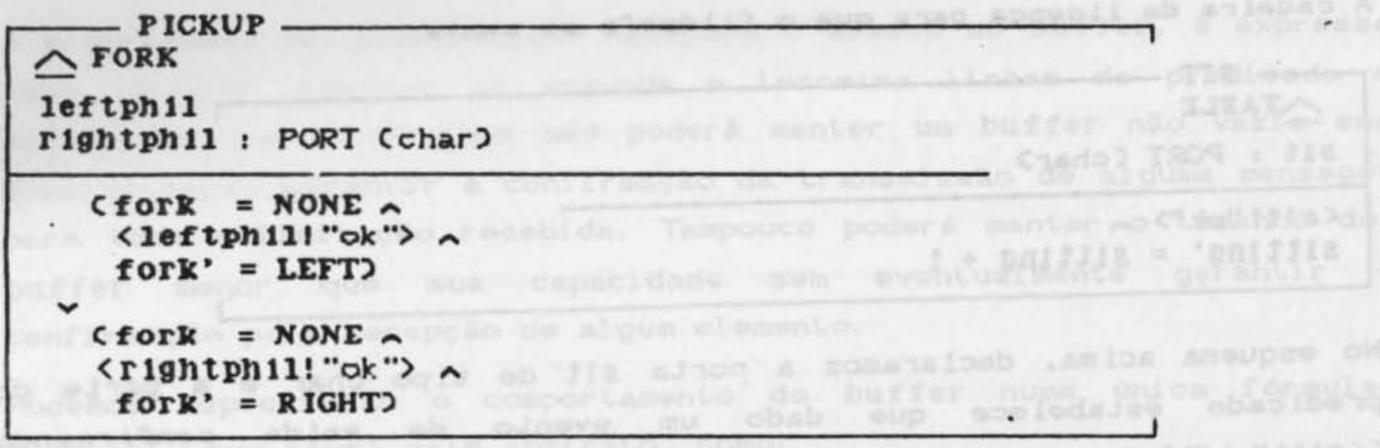
O estado inicial de FORK será NONE para indicar um garfo disponível.

$$FORK\text{-}INITIAL \triangleq [FORK \mid fork = NONE]$$

Os eventos sobre FORK mudam seu estado, limitado por

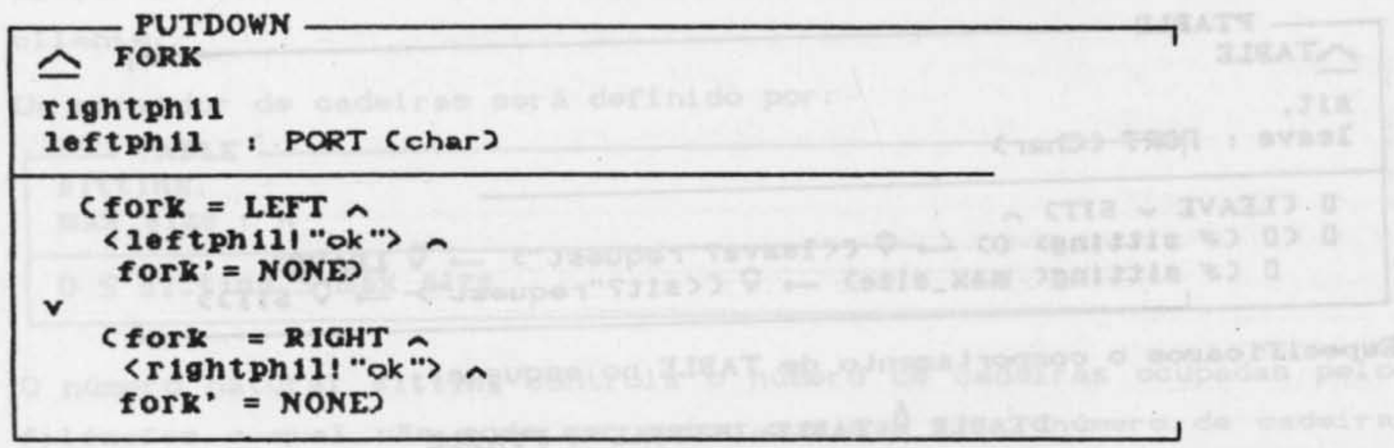
$\triangle FORK \triangleq FORK \wedge FORK'$

As transições dos garfos são especificadas pelos seguintes esquemas:



```
—— PICKUP ——————————————————————
  ∆ FORK
  leftphil
  rightphil : PORT (char)
  ——————————————————————
  (fork = NONE ∧
   <leftphil!"ok"> ∧
   fork' = LEFT)
  ∨
  (fork = NONE ∧
   <rightphil!"ok"> ∧
   fork' = RIGHT)
```

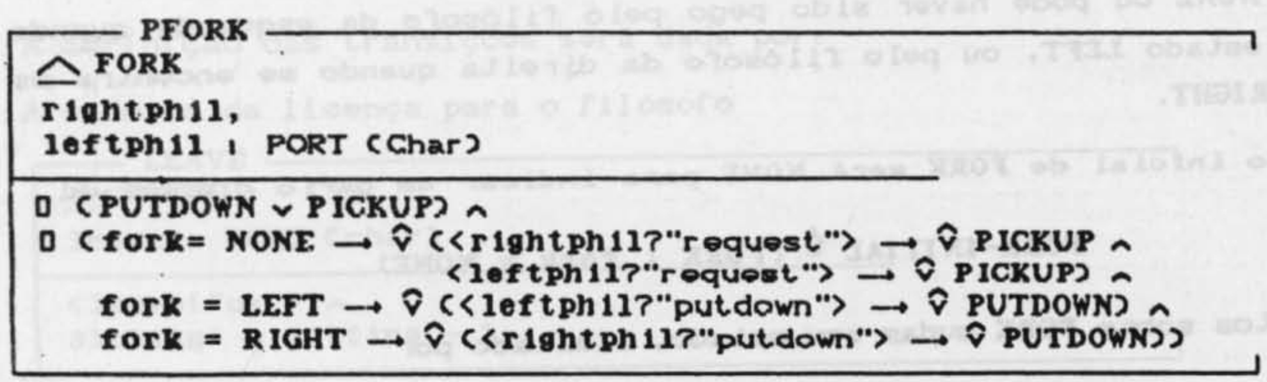
O garfo é pego pelo filósofo da esquerda ou direita. Estando **fork** em estado NONE a ocorrência de um evento de saída confirmando em **leftphil** ou **rightphil** mudará seu estado para LEFT ou RIGHTconforme o caso.



```
—— PUTDOWN ——————————————————————
  ∆ FORK
  rightphil
  leftphil : PORT (char)
  ——————————————————————
  (fork = LEFT ∧
   <leftphil!"ok"> ∧
   fork'= NONE)
  ∨
  (fork = RIGHT ∧
   <rightphil!"ok"> ∧
   fork' = NONE)
```

O garfo é liberado pelo filósofo da esquerda ou direita. O predicado estabelece que o filósofo que tem o garfo é quem pode para liberá-lo. A confirmação do pedido muda o estado de **fork** para NONE, e o garfo está novamente disponível.

Especificamos as propriedades de um garfo no esquema:



```
—— PFORK ——————————————————————
  ∆ FORK
  rightphil,
  leftphil : PORT (Char)
  ——————————————————————
  □ (PUTDOWN ∨ PICKUP) ∧
  □ (fork= NONE → ◇̂ (<rightphil?"request"> → ◇̂ PICKUP ∧
                    <leftphil?"request"> → ◇̂ PICKUP) ∧
     fork = LEFT → ◇̂ (<leftphil?"putdown"> → ◇̂ PUTDOWN) ∧
     fork = RIGHT → ◇̂ (<rightphil?"putdown"> → ◇̂ PUTDOWN))
```

A propriedade de justiça da especificação estabelece que o sistema não poderá manter o fork em estado livre nem nos estados LEFT ou RIGHT indefinidamente sem eventualmente receber algum pedido de leftphil ou rightphil e todo pedido terá uma resposta garantida provocando uma transição de estado.

O comportamento de FORK será definido pelo esquema:

DFORK ≙ FORK-INITIAL ∧ PFORK

Modelamos um filósofo por seus possíveis estados

```
── PHILOSOPHER ──────────────────────────────────
│ philosopher : THINKING | SITTING | EATING
└────────────────────────────────────────────────
```

O filósofo inicialmente vai estar pensando

PHILOSOPHER_INITIAL ≙ [PHILOSOPHER | philosopher = THINKING]

A cada movimento que realiza, um filósofo muda seu estado

ΔPHILOSOPHER ≙ PHILOSOPHER ∧ PHILOSOPHER'

O filósofo pode se sentar

```
── SIT ──────────────────────────────────────────
│ Δ PHILOSOPHER
│ sittable : PORT (char)
├────────────────────────────────────────────────
│ <sittable?"ok"> ∧
│ philosopher = THINKING ∧
│ philosopher'= SITTING
└────────────────────────────────────────────────
```

As linhas do predicado especificam que um evento de entrada confirmando que o filósofo pode se sentar (<sittable?"ok">) faz mudar o estado do philosopher de THINKING para SITTING.

Similarmente definimos as outras possíveis mudanças de estados do filósofo. O filósofo pega os garfos de sua esquerda e direita

```
── EAT ──────────────────────────────────────────
│ Δ PHILOSOPHER
│ leftfork
│ rightfork : PORT (char)
├────────────────────────────────────────────────
│ <leftfork?"ok"> ∧
│ <rightfork?"ok"> ∧
│ philosopher = SITTING ∧
│ philosopher'= EATING
└────────────────────────────────────────────────
```

PDF compressor

e libera os garfos e a cadeira

```
┌── RELEASE ─────────────────────────────────────
│ ^ PHILOSOPHER
│ leavetable
│ leftfork
│ rightfork : PORT (char)
├────────────────────────────────────────────────
│ <leavetable?"ok"> ^
│ <leftfork?"ok"> ^
│ <rightfork?"ok"> ^
│ philosopher = EATING ^
│ philosopher' = THINKING
└────────────────────────────────────────────────
```

As propriedades do filósofo serão expressas por:

```
┌── PPHILOSOPHER ────────────────────────────────
│ ^ PHILOSOPHER
│ sittable,
│ leavetable,
│ leftfork,
│ rightfork : PORT (char)
├────────────────────────────────────────────────
│ □ (SIT v EAT v RELEASE) ^
│ □(philosopher = THINKING ^ <sittable!"request"> → ◊ SIT ^
│   philosopher = SITTING  ^ <leftfork!"pickup"> ^
│                            <rightfork!"pickup"> → ◊ EAT ^
│   philosopher = EATING   ^ <leavetable!"request">^
│                            <leftfork!"putdown"> ^
│                            <rightfork!"putdown"> → ◊ RELEASE)
└────────────────────────────────────────────────
```

Onde a propriedade de justiça assegura que o filósofo não permanecerá muito tempo num mesmo estado mandando mensagens sem eventualmente mudar de estado.

O comportamento de PHILOSOPHER será dado pelo esquema:

DPHIL $\hat{=}$ PHILOSOPHER-INITIAL ^ PPHILOSOPHER

4. ESPECIFICAÇÃO DA ESTRUTURA

Compomos a especificação do comportamento do sistema num esquema chamado estrutura do projeto nos métodos de desenvolvimento top-down[2].

A especificação da estrutura, no caso do exemplo dos filósofos será dado pelo seguinte esquema:

```
── DFAMI ──────────────────────────────────────────────
  phil  : seq DPHIL
  fork  : seq DFORK
  table : DTABLE
───────────────────────────────────────────────────────
  # fork = # phil
  table.max_size < # phil
  (∀ k: 1..# phil•
     phil(k).sittable      <<>>  table.sit
     phil(k).leavetable    <<>>  table.leave
     phil(k).leftfork      <<>>  fork(k).rightphil
     phil(k).rightfork     <<>>  fork(MOD(k,#fork)+1).leftphil)
───────────────────────────────────────────────────────
```

O conector <<>> tem a seguinte semântica informal: s.w <<>> t.q especifica a conexão da porta w do modulo s com a porta q do t. Quando duas portas são conectados, elas são intercalados numa única porta idêntica: qualquer evento para uma é evento para a outra.

Este esquema declara **phil**, uma seqüência de DPHIL, **fork**, uma seqüência de DFORK, e table, como DTABLE. O predicado especifica que o número de elementos de **phil** deve ser igual ao de **fork** (número de filósofos igual o número de garfos), que **max_size** de **table** deve ser menor que o número de elementos de **phil** (evita o deadlock) e descreve uma série de conexões entre as portas de **phil** com os de **fork** e **table**.

5. VANTAGENS E CONCLUSÕES

Lógica temporal tem demostrado ser uma ferramenta útil no raciocinio sobre o comportamento de programas concorrentes[5]. Noso método difere de outros[1,4] porque permite especificar o comportamento e a estrutura do sistema, não deixa implícito as propriedades de segurança e justiça e porque requer a generalização de lógica temporal para incluir predicados sobre esquemas, os quais descrevem as ações que são executadas e logram abstraer operações indivisíveis.

A linguagem desenvolvida aqui permite especificar sistemas distribuídos com as seguintes vantagens:

- Modularidade: Onde propriedades distintas podem ser especificadas independentemente, o que permite uma especificação mínima, que pode ser estendida, controlando a complexidade do processo de verificação.
- Formalidade: O cálculo de predicados e a lógica modal são teorias matemáticas fortes.
- Generalidade: As propriedades de segurança, justiça, as relacionadas aos dados e ao controle podem ser especificadas e verificadas.
- Exatidão: Interferência entre processos é controlada representando as operações como ações atômicas e restringindo sua execução ao

PDF compressor

cumprimento de certas asserções. Concorrência é modelada pela intercalação de ações atômicas.

- Facilidade de Uso: A notação matemática usada permite especificar propriedades numa forma abstrata (não orientada a uma implementação em particular), permite a utilização de técnicas de refinamento, e não exclui o uso da linguagem natural.

Esta linguagen será formalmente especificada para a construção de ferramentas de tratamento.

Esta Notação está sendo considerada como ferramenta para especificação de sistemas que são naturalmente reativos ou que vão ter implementação de forma distribuida.

6. BIBLIOGRAFIA

[1] A. Pnueli : Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Trends, Department of Applied Mathematics The Weizmann Institute of Science, Rehovot 76100, Israel.

[2] Bo-Shoe Chen: Formal Specification and Verification of Distributed Systems, IEEE Trans. Software Engineering SE-9 (1983), 710-730.

[3] I. Hayes (ed): Specification Case Studies, Prentice Hall Intl.(UK), 1987.

[4] L. Lamport: Specifying Concurrent Program Modules, TOPLAS 5,2 (1983), 190-222.

[5] L. Lamport: What good is temporal logic?, Proceedings IFIP, (1983), 657-668.

[6] S. Owicki, L. Lamport: Proving Liveness Properties of Concurrent Programs, ACM TOPLAS 4,3 (1982), 455-495.

This document was created with an evaluation copy of CVISION PDFCompressor