UMA ANALISE DE ADEQUABILIDADE E VALIDAÇÃO DE PRIMITIVAS DE COMUNICAÇÃO ENTRE PROCESSOS

Augusto César Alves Sampaio
Paulo Roberto Freire Cunha

Departamento de Informática
Universidade Federal de Pernambuco
50739, Recife, PE, BRASIL

RESUMO

Apresentamos algumas considerações gerais acerca dos mais variados conceitos e notações que têm sido propostos para interação entre processos e descrevemos as primitivas de comunicação de algumas linguagens (CSP, Ada e CONIC), para ilustrar implementações de algumas das notações com suas respectivas semânticas. Escolhemos Ada e CONIC, que são representativas de uma grande classe de linguagens concorrentes, para efetuar uma análise de adequabilidade e validação de suas primitivas, o que é feito de forma comparativa.

1. INTRODUÇÃO

Processos concorrentes normalmente se comunicam, a fim de cooperarem para a execução da tarefa como um todo. Existem duas formas de comunicação entre processos: o uso de variáveis compartilhadas e através de passagem de mensagens.

Quando variáveis compartilhadas são usadas para comunicação entre processos, dois tipos de sincronização se verificam: exclusão mútua e sincronização por condição. Exclusão mútua assegura que uma seqüência de comandos seja tratada como uma operação indivisível. Esta seqüência de comandos é denominada região crítica. Na sincronização por condição, se uma dada variável contém um valor inapropriado para a execução de uma operação particular, o processo deve esperar até que um outro processo altere a variável para um valor apropriado à sua execução.

Passagem de mensagens pode ser vista como uma técnica tanto para transferir dados como para implementar sincronização entre processos.

Quando passagem de mensagens é usada para comunicação e sincronização, processos enviam e recebem mensagens ao invés de utilizarem variáveis compartilhadas.

A seção 2 deste trabalho descreve algumas considerações acerca das primitivas de comunicação e sincronização baseadas em variáveis compartilhadas e passagem de mensagens. Algumas linguagens são descritas na seção 3, a fim de ilustrar implementações reais das principais construções descritas na seção 2. A linguagem CSP [Hoare 78] é utilizada para ilustrar uma solução simples e elegante de comunicação entre processos. Grande parte dos conceitos incorporados em CSP são originais e foram, de certa forma, o ponto de partida para o desenvolvimento de primitivas mais poderosas que se sucederam. Ada [Pyle 81, USDoD 81] e CONIC [Dulay 84, Kramer 81, Kramer 84, Sloman 85] são representativas de uma grande classe de linguagens de programação concorrente. Ada suporta aplicações de tempo real e tem sido amplamente adotada internacionalmente. CONIC é uma ferramenta relativamente nova e apresenta conceitos inovadores que facilitam o desenvolvimento de sistemas distribuídos e, em particular, aplicações de controle de processos.

Análise de adequabilidade e validação das primitivas disponíveis nas diversas linguagens de programação concorrente continua sendo tema de interesse, desde que estão sempre surgindo novas construções que precisam ser investigadas. Paralelamente à descrição das primitivas, analisamos o potencial de comunicação oferecido por Ada e CONIC, de forma comparativa. Alguns aspectos considerados incluem as classes de aplicações melhor suportadas por cada linguagem; as formas de compartilhamento de variáveis oferecidas; aspectos sintáticos e operacionais (semânticos) das primitivas de chamada a procedimento remoto, onde discutimos as facilidades e dificuldades oferecidas para a validação de aplicações desenvolvidas nas duas linguagens; e a designação dos canais de comunicação.

Finalmente, apresentamos algumas considerações gerais a respeito do artigo e citamos alguns trabalhos que serão desenvolvidos a curto e médio prazos, como uma continuação da experiência adquirida durante nossa análise de adequabilidade e validação de primitivas de comunicação entre processos.

2. PRIMITIVAS DE COMUNICAÇÃO ENTRE PROCESSOS

Diversas construções baseadas em variáveis compartilhadas têm sido propostas para implementar sincronização entre processos, como os semáforos de Dijkstra [Djkst 68], região crítica condicional [Hoare 72], monitor [Hoare 74] e expressões de caminho ("path expressions") [Campbe 76]. Contudo, por serem baseadas no uso de variáveis compartilhadas, tais mecanismos não atendem à necessidade atual de sistemas "completamente" distribuídos.

Primitivas baseadas em passagem de mensagens oferecem uma maior naturalidade para a interação entre processos. Quando se utiliza passagem de mensagens, comunicação significa que um processo, ao receber uma mensagem, obtém valores do processo que a enviou. Sincronização refere-se ao fato de que uma mensagem só pode ser recebida se ela foi enviada.

Várias construções com variadas notações sintáticas e diferenças conceituais (semânticas) têm sido propostas para passagem de mensagens, desde as construções mais primitivas como "send" e "receive" até construções de mais alto nível, tais como chamada a procedimento remoto e transações atômicas.

Naturalmente, existem várias semânticas que podem ser associadas a construções sintáticas idênticas, mesmo para as primitivas mais simples como "send" e "receive". Dois principais aspectos a serem decididos na implementação destas primitivas caracterizam as variações de seus significados: o primeiro, refere-se à maneira como a origem e o destino são designados (como o canal de comunicação é especificado) e o segundo, ao modo que a comunicação entre estas primitivas é sincronizada (com bloqueio ou não).

2.1 Especificação do canal de comunicação

O método mais simples de se especificar o canal de comunicação é através da designação direta do destino e da origem com o nome dos processos envolvidos na comunicação. Este método é simples de implementar e utilizar, porém não é adequado para aplicações como

cliente/servidor onde se deseja referenciar a clientes e servidores genéricos.

Uma solução para este problema é designar a origem e o destino através de nomes globais, também chamados de "mailbox", onde as mensagens enviadas para um dado "mailbox" podem ser recebidas por qualquer processo que execute um "receive" cuja origem é tal "mailbox". Porém, a implementação de "mailbox" é bem mais complexa do que designação direta.

Um caso especial de nomes globais, no qual um "mailbox" pode aparecer como designador de origem em um único processo, oferece uma restrição a nomes globais (desde que vários clientes podem enviar para o mesmo servidor, mas um cliente não pode enviar para servidores genéricos), porém, é mais simples de implementar. Este tipo de "mailbox" é denominado de portas.

2.2 Sincronização

Quanto à sincronização, diferenças na implementação referem-se ao fato do processo permanecer ou não bloqueado quando executando uma primitiva. No caso da primitiva de envio de mensagens ser assíncrona, a mensagem enviada pode tornar-se obsoleta, desde que quando recebida pode já não refletir o atual estado do processo que a enviou. Por outro lado, se não há espaço para armazenar as mensagens enviadas, o "send" (ou similar) deve ser necessariamente síncrono. Neste caso, a mensagem recebida sempre corresponde ao estado atual do processo que a enviou.

As considerações para o receptor são semelhantes, porém, normalmente a implementação da primitiva de recepção é síncrona. As linguagens geralmente oferecem meios para um "receive" não bloqueado (a fim de evitar um bloqueio infinito), como também, a possibilidade de selecionar uma, entre várias mensagens a serem recebidas. Algumas construções que possibilitam seleção na recepção de mensagens são descritas a seguir.

O comando

receive lista-de-variáveis from origem when b

permite apenas receber as mensagens para as quais a expressão booleana

"b" retorna uma valor verdadeiro. PLITS [Andrew 83] e SR [Andrew 83] oferecem tais facilidades.

Uma forma mais genérica foi proposta por Dijkstra [Dijkst 75], com o uso de expressões de guarda

```
if   G1 --> S1
   □G2 --> S2
...
   □Gn --> Sn
fi
```

Inicialmente, todas as expressões de guarda ("G1" a "Gn") são avaliadas. As que retornarem valor verdadeiro fazem com que os respectivos comandos de passagem de mensagens (que fazem parte das expressões de guarda) sejam elegíveis. Em seguida, só para as guardas elegíveis, é verificado se existem mensagens disponíveis para serem recebidas. Então, um destes comandos de passagem de mensagens em "Gi" (se presente) é arbitrariamente executado. Finalmente, o comando "Si" associado é executado. Contudo, se nenhuma mensagem estiver pronta para ser recebida, o processo é suspenso até que uma mensagem se torne disponível. Note que isto pode causar um bloqueio infinito.

2.3 Construções de passagem de mensagens de mais alto nível

Embora as primitivas "send" e "receive" sejam suficientes para programar qualquer tipo de interação entre processos usando passagem de mensagens, existem problemas cujo uso destas primitivas não oferecem uma solução prática, como, por exemplo, para problemas genéricos do tipo cliente/servidor.

2.3.1 Chamada a procedimento remoto

Chamada a procedimento remoto suporta o tipo de interação cliente/servidor através da execução de uma única primitiva, cuja sintaxe geral é

call serviço (<parâmetros argumento>; <parâmetros resultado>)

onde serviço é um nome de canal, podendo ser uma designação direta, uma porta ou um nome global.

A execução do "call" é semelhante a uma chamada a procedimento numa linguagem seqüencial: os <parâmetros argumento> são enviados ao servidor designado e o cliente fica bloqueado até que o serviço tenha sido realizado, quando os resultados são atribuídos aos <parâmetros resultado>.

Há duas abordagens principais para especificar o servidor. Na primeira, o procedimento remoto é uma declaração, da mesma forma que numa linguagem seqüencial.

```
remote procedure serviço (<parâmetros argumento>;
                          <parâmetros resultado>)

    corpo
end
```

Contudo, tal declaração de procedimento é implementada como um processo. Este processo espera receber uma mensagem contendo argumentos do processo que solicitou o serviço, os quais são atribuídos aos seus <parâmetros argumento>. Então, seu corpo é executado, retornando uma mensagem ("reply") contendo os valores dos resultados (se houver). Na realidade, os resultados são atribuídos aos <parâmetros resultado>, o "reply" é implícito.

Na segunda abordagem, o procedimento remoto é implementado através de um comando. Tal comando tem a seguinte forma geral:

```
accept serviço (<parâmetros argumento>; <parâmetros resultado>)
    --> corpo
end
```

Quando um "accept" (ou comando semelhante) é usado para especificar o lado do servidor, a chamada a um procedimento remoto é denominada "rendezvous", pois há um "encontro" entre o cliente e o servidor durante a execução do corpo do "accept" que, então, seguem caminhos separados. Uma importante vantagem deste enfoque é que o servidor pode fornecer vários serviços, onde cada "accept" implementa um tipo de serviço.

2.3.2 Transações atômicas

Chamada a procedimento remoto oferece condições favoráveis a uma programação clara da interação entre processos, porém, há outros fatores a considerar, como, por exemplo, a tolerância a falhas. Durante

o "rendezvous" entre dois processos, pode ocorrer uma falha no servidor e não é desejável que o cliente permaneça bloqueado indefinidamente.

Uma solução é associar, ao cliente, um intervalo de tempo. Se nenhuma resposta for recebida pelo cliente antes que o intervalo de tempo expire, o cliente assume que houve falhas no servidor e realiza alguma ação. O problema desta solução é decidir qual ação deve ser tomada, pois não é simples descobrir a causa da falha. (Perda da solicitação do serviço, perda da resposta ou o servidor entrou em "loop" durante o "rendezvous"?).

Uma solução mais geral, porém, é considerar a execução de um procedimento remoto em termos de transações atômicas. Uma transação atômica é uma computação do tipo "tudo ou nada", ou seja, o estado do processo só é alterado se o serviço solicitado foi completo, caso contrário, nenhuma modificação de estado é efetuada. Esta característica denomina-se atomicidade a falhas. Além disso, transações atômicas são indivisíveis no sentido de que a execução parcial de uma transação atômica não é visível a qualquer transação atômica em execução concorrente. Este atributo é denominado atomicidade à sincronização.

A principal contribuição de transações atômicas é o fato de impedir que um processo visualize dados do sistema num estado inconsistente. Porém, a implementação de transações atômicas é complexa, envolvendo um custo alto.

3. ALGUMAS LINGUAGENS DE PROGRAMAÇÃO CONCORRENTE

Nesta seção, descrevemos brevemente as primitivas de sincronização de algumas linguagens de programação concorrente (CSP, Ada e CONIC) e as associamos com os conceitos e notações apresentados na seção anterior. Para situar o leitor acerca das linguagens que descrevemos, com relação a um âmbito geral, apresentamos uma classificação para linguagens concorrentes proposta em [Andrew 83].

3.1 Uma classificação para linguagens concorrentes

Podemos agrupar as linguagens em três classes: orientadas a

procedimentos, orientadas a mensagens e orientadas a operações. Linguagens numa mesma classe fornecem os mesmos mecanismos básicos de interação e têm atributos similares.

Em linguagens orientadas a procedimentos, a interação entre processos é baseada em variáveis compartilhadas. Portanto, linguagens orientadas a procedimentos normalmente fornecem meios para garantir exclusão mútua, como é o caso de Pascal Concorrente, Módula e Mesa, dentre outras.

Linguagens orientadas a mensagens e a operações são ambas baseadas em passagem de mensagens, mas refletem diferentes aspectos da interação entre processos. As orientadas a mensagens fornecem "send" e "receive" (ou notações similares) como o principal meio de interação entre processos. Em contraste com as linguagens orientadas a procedimentos, não há variáveis compartilhadas. Como exemplo, temos CSP, Gypsy e PLITS, dentre outras.

As linguagens orientadas a operações oferecem chamada a procedimento remoto como o principal meio para interação entre processos. Estas linguagens combinam aspectos das outras duas classes, ou seja, tanto há interação por passagem de mensagens, como por compartilhamento de variáveis. Nesta classe, temos Ada, SR, Distributed Processes e CONIC, dentre outras.

3.2 CSP

Em CSP, os processos podem compartilhar variáveis, porém, só de leitura. Comunicação e sincronização são fornecidas através de comandos de entrada/saída (para receber e enviar mensagens, respectivamente), cujas sintaxes definimos a seguir.

```
destino!expressão

origem?variável
```

As semânticas correspondem respectivamente ao "send" e "receive", onde destino e origem são designações diretas de canais e tanto o comando de saída ("!") como o de entrada ("?") são síncronos.

CSP oferece uma forma restrita de comando de seleção de comunicação. Comandos de entrada podem aparecer em expressões de guarda

de comandos de alternação ou repetição, mas os de saída não podem.

A designação de canais em CSP é direta, ou seja, destino e origem são nomes de processos. Como a comunicação é sincrona, não há alocação automática de "buffers" para as mensagens enviadas. Esta questão do processo que envia a mensagem ficar ou não bloqueado até que o processo destino a receba é um tanto polêmica e depende da aplicação. Se é importante para uma dada aplicação (por exemplo, aplicações de tempo real) que o processo receptor conheça o atual estado do processo transmissor, é evidente que o transmissor deve ficar bloqueado até que o outro processo receba a mensagem. Por outro lado, há aplicações que não justificam o bloqueio do processo transmissor, como apresentado na descrição da transação "notify" de CONIC (seção 3.4).

3.3 Ada

Com respeito à programação concorrente, a principal inovação de Ada é a forma de chamada a procedimento remoto denominada "rendezvous". Processos em Ada são chamados de tarefas. Tarefas podem ser declaradas dentro de outras tarefas e podem interagir através do uso de variáveis compartilhadas declaradas em blocos mais externos. A garantia de acesso exclusivo é fornecida pela procedure "shared_variable_update". As tarefas são seqüenciais.

A chamada a procedimento remoto é feita através do comando "call". Os procedimentos remotos são chamados de "entries", que são portas dentro de um processo servidor. Portanto, aplicações do tipo vários clientes e um único servidor são diretamente suportadas, o que não acontece para aplicações onde se têm vários servidores. A recepção de mensagens nas portas é especificada através do comando "accept". Para ilustrar o uso do "call" e "accept" em Ada, um exemplo de um processo servidor que tem a função de agir como o canal de comunicação, no qual caracteres podem ser escritos e lidos por diferentes tarefas é apresentado a seguir.

```
task unico_buffer is
  entry write (ch: in character);
  entry read  (ch: out character);
end unico_buffer;
```

```
task body unico_buffer is
  char: character;
  begin
    loop
      accept write (ch: in character) do
        char:= ch;
      end write;
      accept read (ch: out character) do
        ch:= char;
      end read;
      exit when char = ascii.eot;
    end loop;
  end unico_buffer;
```

Os dois serviços "write" e "read" devem ser usados alternadamente (só um caracter é armazenado por vez). Para cada "entry" tem que existir pelo menos um "accept" associado. Observe que os <parâmetros argumento> e <parâmetros resultado> descritos no item 2.3.1 correspondem em Ada a "in" e "out", respectivamente.

Dois processos clientes ("produtor" e "consumidor") são apresentados a seguir, para ilustrar a utilização dos dois serviços oferecidos pelo servidor descrito acima.

```
task produtor;
task body produtor is
  c1: character;
  begin
    loop
      (* gera c1 *)
      unico_buffer.write (c1);
    end loop
  end produtor;

task consumidor;
task body consumidor is
  c2: character;
```

```
begin
  loop
    unico_buffer.read (c2);
    (* uso de c2 *)
  end loop
end consumidor;
```

Observe que a chamada ao serviço é feita pelo nome da tarefa seguida de um ponto (".") e do nome do serviço. Embora sintaticamente diferente do "call" descrito no item 2.3.1, a semântica é a mesma.

O comando "select" permite uma solução na comunicação de forma semelhante ao comando de seleção descrito no item 2.2. Ada dispõe do comando "select" tanto para a tarefa cliente (também denominada ativa) quanto para o servidor (ou passiva).

A sintaxe do "select" para a tarefa ativa (que inicializa a comunicação) é dada por

```
select
  (* chamada a um procedimento *)
else
  (* a tarefa passiva não está pronta para o "rendezvous"*)
end select
```

onde a semântica pode ser facilmente deduzida pelos comentários inseridos no "select" acima.

Uma outra forma do "select" para o procedimento ativo é dada por

```
select
  (* chamada a um procedimento *)
or delay t
  (* comandos *)
end select
```

Estes mecanismos de desvio de bloqueio servem para evitar que o processo ativo permaneça bloqueado indefinidamente à espera de um "rendezvous" que pode não ocorrer. No primeiro caso, se o processo passivo não estiver imediatamente pronto para a comunicação, a parte "else" é satisfeita. No segundo caso, o processo ativo espera por "t"

unidades de tempo. Se neste intervalo o processo passivo executar o "accept" correspondente, o "rendezvous" ocorre, caso contrário, o processo ativo passa a executar os comandos seguintes ao "delay".

Algumas considerações merecem ser destacadas. Por exemplo, o tempo de espera "t" (na segunda forma do "select") refere-se ao tempo até que a tarefa passiva esteja pronta para o "rendezvous", e não ao fim do "rendezvous". Esta semântica associada ao "delay" não oferece um mecanismo que permita ao processo ativo sair do bloqueio para os diversos casos de erros que podem ocorrer durante o "rendezvous", tais como: falha na tarefa passiva ou perda da resposta ao serviço pelo meio de comunicação. Ocorrendo qualquer dos casos, a única possibilidade disponível em Ada para retirar o processo do bloqueio é através da interferência de um terceiro processo, o que é, no mínimo, um tanto arbitrário e, portanto, não oferece uma segurança satisfatória. Outro aspecto refere-se à semântica do "else" (na primeira forma de "select"), que corresponde a um "delay 0" (zero unidades de tempo).

Quanto às tarefas passivas, o "select" é mais flexível, provendo o não determinismo, ou seja, permite que o processo passivo entre em comunicação com um dos possíveis processos ativos que estejam prontos para o "rendezvous", de forma arbitrária.

```
select
  accept ...
or
  accept ...
...
or
  delay t
end select
```

A semântica deste "select" é semelhante ao comando de seleção proposto por Dijkstra, descrito anteriormente. Pode haver mais de um "delay" no mesmo "select", mas, naturalmente, isto corresponde a um único "delay" (o que tem o menor valor "t" como parâmetro). Os comentários feitos quando da descrição da cláusula "delay" do "select" para tarefas ativas também são válidos aqui.

Outra variação possível do "select" é incluir numa das

alternativas ("or") o comando "terminate", que permite que uma outra tarefa cancele a tarefa que contém o "terminate". Normalmente, o "terminate" é utilizado quando o "select" é usado dentro de um comando "loop", para permitir que a tarefa possa ser cancelada entre a execução de duas iterações.

Expressões de guarda semelhantes às descritas anteriormente também são disponíveis em Ada, a fim de permitir uma pré-seleção nas alternativas de um "select", para decidir quais as elegíveis. Da mesma forma, a alternativa "else" também pode ser utilizada, podendo atuar de duas formas: como a negação de todas as expressões de guarda, significando que se nenhuma delas for verdadeira, a parte "else" é executada; e uma segunda, corrrespondendo a um "delay 0", ou seja, se não houver pelo menos uma tarefa imediatamente pronta para o "rendezvous" com um dos comandos "accept" elegíveis, a parte "else" é executada.

Apesar dos problemas operacionais discutidos, as primitivas de Ada têm-se mostrado adequadas para a interação entre processos. A sintaxe das primitivas contribuem para exprimir, numa forma elegante, os seus significados semânticos associados. Além disso, as primitivas são bem estruturadas, facilitando uma programação clara e livre de erros.

3.4 CONIC

O desenvolvimento de um sistema distribuído em CONIC é dividido em duas fases bem distintas: a programação de módulos de software individuais, que se comunicam com o exterior unicamente através de portas de entrada e saída locais ao módulo ("entryports" e "exitports", respectivamente); e uma segunda fase consistindo na configuração destes módulos, onde são criadas instâncias dos módulos e são feitas as devidas interconexões entre "exitports" e "entryports".

Este enfoque de CONIC para o desenvolvimento de sistemas distribuídos constitui a principal contribuição (inovação) da linguagem, pois fornece uma independência de configuração, permitindo que os módulos sejam reusados em várias situações.

Reconfiguração dinâmica (que permite modificações no sistema em

tempo de execução), unidades de definição (que podem ser armazenadas em bibliotecas e utilizadas por diversos módulos) e compilação separada, tanto dos módulos quanto das unidades de definição, são outras características que tornam CONIC uma ferramenta de grande utilidade para a construção de sistemas distribuídos. Uma descrição mais detalhada de tais características está fora do escopo deste artigo, mas pode ser encontrada em [Sloman 85]. Nosso intuito é a descrição e análise das primitivas de comunicação e sincronização. A seguir, apresentamos o mesmo exemplo do "produtor-consumidor" em CONIC.

```
task module unico_buffer
 entryport write: character reply signaltype
           read : signaltype reply character;
  var char: character;
  begin
    loop
      receive char from write reply signal;
      receive signal from read reply char;
      if char = ascii.eot then exit;
    end
  end. (unico_buffer)


task module produtor;
  exitport write: character reply signaltype;
  var c1: character;
  begin
    loop
      (* gera c1 *)
      send c1 to write wait signal;
    end
  end. (produtor)

task module consumidor;
  exitport read: signaltype reply charater;
  var c2: character;
begin
  loop
```

```
        send signal to read wait c2;
        (* uso de c2 *)
      end;
    end. (consumidor)
```

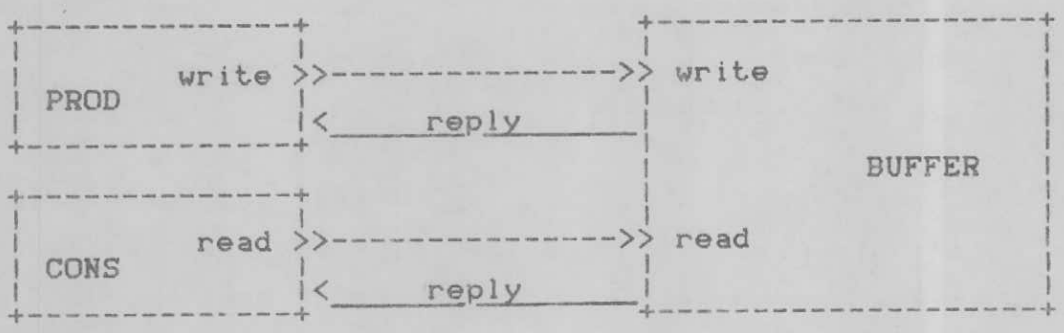
De forma semelhante à Ada, as tarefas são seqüencias, porém, pode ser facilmente observado que, devido à separação entre programação e configuração, as tarefas apresentadas acima são completamente independentes. Portanto, na programação de uma tarefa, não existe qualquer referência a outras.

Na realidade, as tarefas como estão definidas não representam ainda o sistema que pretendemos desenvolver, pois podem ser considerados como tipos de dados que precisam ser instancializados e interconectados no nível de configuração, conforme descrito a seguir.

```
system prod_cons;
  use unico_buffer, produtor, consumidor;
  create buffer: unico_buffer;
         prod: produtor;
         cons: consumidor;
  link prod.write to buffer.write
       cons.read  to buffer.read
end
```

A construção "use" identifica os tipos de módulos a partir dos quais o sistema é construído. Os nomes das intâncias dos módulos dentro do sistema são declarados pela construção "create". As interconexões das portas (dos módulos já instanciados) são declaradas através da construção "link".

Várias instâncias do mesmo módulo podem ser definidas, inclusive por sistemas diferentes, com finalidades distintas. A única restrição é que as portas que estão sendo conectadas têm que possuir o mesmo tipo. Observe que as portas "read" e "write" têm os mesmos nomes (nas tarefas "consumidor" e "produtor" ,respectivamente) da tarefa "unico_buffer" por simples conveniência, pois poderiam ser distintos. A figura abaixo ilustra uma visão do sistema do ponto de vista de configuração.



Embora não utilize nomes globais, CONIC permite uma comunicação multi-destino, pois as portas através das quais os processos se comunicam são declaradas tanto do lado do servidor como do cliente. Portanto, em um par de processos que se comunicam, um não sabe quem é o outro. Isto é possível devido à separação entre programação e configuração, talvez a principal característica de CONIC.

Assim, é possível que uma "exitport" de um dado processo seja conectada a várias "entryports" de processos distintos (em tempo de configuração). Então, quando uma mensagem é enviada à "exitport", todos os processos cujas "entryports" foram conectadas receberão a mensagem.

Aplicações do tipo um servidor e vários clientes são implementadas de modo análogo, só que várias "exitports" são ligadas a uma única "entryport" do processo servidor. Ainda é possível a implementação do caso genérico de vários servidores e vários clientes, onde a "entryport" de cada servidor é conectada às "exitports" dos diversos clientes. Naturalmente, a comunicação do tipo multi-destino só faz sentido para as transações onde o processo ativo não fica bloqueado ao enviar uma mensagem (denominadas em CONIC de "notify").

Portanto, CONIC oferece uma flexibilidade considerável em relação à Ada quanto à especificação do canal de comunicação. Mais ainda, a solução proposta é simples, modular e simétrica, desde que a programação das tarefas é exatamente a mesma para quaisquer das aplicações discutidas acima, pois a conexão entre as portas é feita no nível de configuração.

CONIC permite a declaração de dois tipos de portas que correspondem às classes de transações de mensagens existentes. Portas

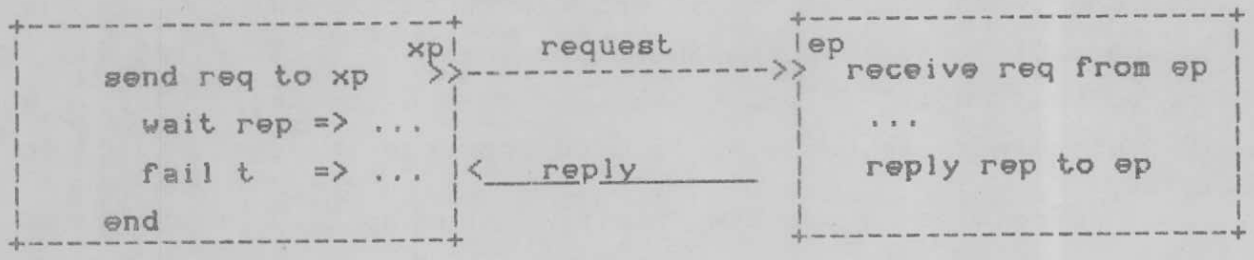
"request-reply", tais como as declaradas no exemplo do "produtor-consumidor", são bidirecionais, especificando um pedido e uma resposta. Esta transação é do tipo chamada a procedimento remoto, só que a parte da resposta ("reply") é explícita. Obviamente, portas do tipo "request-reply" são síncronas tanto do lado do processo ativo quanto do passivo.

O outro tipo de porta é "notify", que não tem a parte "reply". As primitivas de comunicação para portas do tipo "notify" são "send", para enviar uma mensagem por uma "exitport", e "receive", para receber uma mensagem numa "entryport". Neste tipo de transação o "send" é assíncrono e o "receive" é síncrono. As mensagens devem possuir os tipos correspondentes aos tipos das portas, o que é verificado em tempo de compilação.

Em Ada, a única maneira possível de passagem de mensagens é através de "rendezvous", onde ambos os processos ficam bloqueados até o término da interação, mesmo que nenhum parâmetro seja efetivamente retornado do processo passivo para o ativo.

Há aplicações onde a tarefa ativa deseja apenas fornecer informações de status à tarefa passiva, como, por exemplo, tarefas que efetuam leitura de sensores, temperatura, pressão, nível da água, etc. Portanto, CONIC fornece, neste sentido, uma flexibilidade maior que Ada, desde que para tais aplicações, não há necessidade de que o processo ativo permaneça bloqueado até que o receptor esteja pronto para receber a mensagem. Para aplicações onde é importante que as informações não se percam, a transação "request-reply" deve ser usada, desde que a parte "reply" confirma o recebimento da mensagem.

As primitivas para portas do tipo "request-reply" são semelhantes, só que incluem uma construção para especificar a parte "reply" e (no caso do processo ativo) mecanismos para evitar um bloqueio infinito.



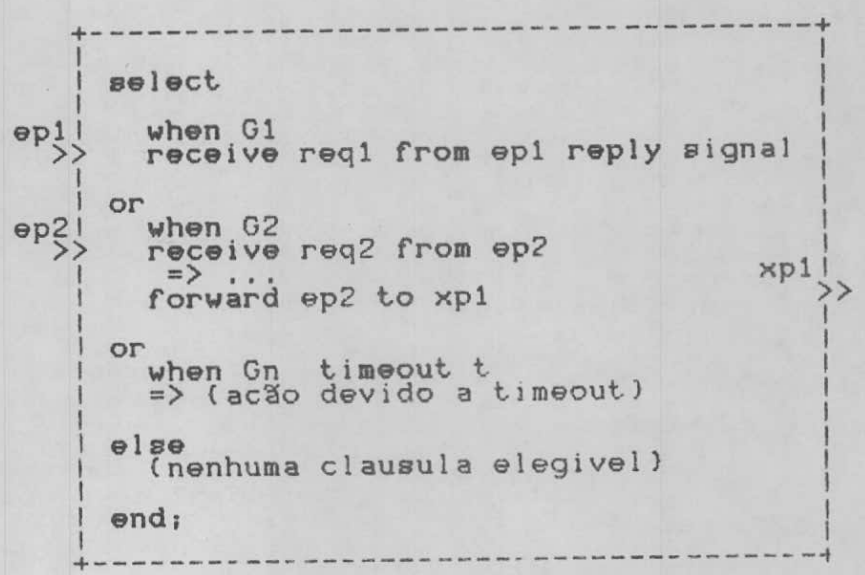
Num processo ativo usando uma porta do tipo "request-reply", a parte "reply" é especificada através da cláusula "wait", indicando que a tarefa ficará bloqueada até a resposta seja enviada pela tarefa passiva, através da cláusula "reply".

A cláusula "fail" na tarefa ativa serve como um mecanismo para evitar que o processo fique bloqueado indefinidamente. O parâmetro "t" indica as unidades de tempo que o processo permancerá bloqueado até a resposta ser recebida. Caso o tempo expire, o processo sai do bloqueio e toma alguma decisão.

Após receber a mensagem do processo ativo, o processo passivo pode executar outros comandos antes de enviar o "reply". Ao invés de enviar o "reply" ao processo do qual recebeu a mensagem, o receptor pode redirecionar o pedido a outro processo (e assim por diante), através da cláusula "forward". Neste caso, o último processo a receber a mensagem deve enviar o "reply". Outra possibilidade refere-se ao fato de que o receptor pode abortar a transação, caso em que a cláusula "fail" (se houver), no processo ativo, será executada.

Como não há uma garantia de que uma "exitport" de uma dada instância de um módulo tarefa está conectada a uma "entryport" de outro módulo, CONIC oferece funções padrão que determinam se a ligação existe, o número de mensagens prontas a serem recebidas numa dada "entryport" e a razão da falha ocorrida quando a cláusula "wait" é executada.

A figura abaixo ilustra um exemplo geral do comando "select" disponível na linguagem CONIC.



Qualquer combinação das primitivas "receive-reply", "receive-forward" ou "receive-abort" podem ser usadas num comando "select", a fim de fornecer não-determinismo associado a alguns mecanismos que evitam o processo receptor permanecer bloqueado indefinidamente. Primitivas de envio de mensagem não podem ser utilizadas no "select". A sintaxe e semântica não apresentam diferenças consideráveis em relação ao comando de seleção de Ada. Algumas considerações operacionais sobre a transação "request-reply" e o comando "select" são discutidas a seguir.

De forma geral, podemos considerar a transação "request-reply" disponível em CONIC como um melhoramento do "rendezvous" de Ada. Analisando apenas as primitivas, vemos que apesar da diferença sintática existente, elas possuem exatamente a mesma semântica. As divergências ocorrem quanto aos mecanismos utilizados para evitar que os processos permaneçam bloqueados eternamente. Consideremos primeiro o processo ativo. A cláusula "delay" associada ao comando "select" de Ada limita o tempo que o processo ativo permanece bloqueado até que o processo passivo correspondente esteja pronto para o "rendezvous", o que pode ocasionar os problemas discutidos anteriormente.

Em CONIC, a cláusula "fail" associada ao "send" limita o tempo que o processo ativo permanece bloqueado até o término da transação "request-reply", ou seja, quando a tarefa passiva executa a parte "reply" e a resposta é recebida pela tarefa passiva. Este mecanismo

torna a implementação do temporizador mais simples do que em Ada, desde que seu efeito é local à tarefa ativa. Portanto, não há necessidade do sistema de comunicação cancelar uma requisição de serviço pendente, o que ocorre em Ada quando a cláusula "delay" é executada. Contudo, em CONIC, as respostas enviadas para as requisições onde ocorreram "estouro" do temporizador devem ser descartadas.

Além disto, a função "reason" (usada dentro da cláusula "fail") permite à tarefa ativa identificar a causa da falha: retorna um dado valor se a transação foi abortada pelo receptor, pelo meio de comunicação ou sistema operacional; outro valor se ocorrer um estouro do temporizador e, um outro valor para indicar que a porta de saída ("exitport") em questão não estava conectada a qualquer "entryport".

Portanto, CONIC oferece um mecanismo de evitar que o processo ativo permaneça bloqueado mais poderoso e intuitivo do que o disponível em Ada, pois apesar de não oferecer transações atômicas, permite condições de detectar a causa da falha de comunicação, através da função "reason".

Consideremos, agora, a tarefa passiva. A sintaxe e semântica do comando "select" utilizada para a recepção de mensagens são bastante semelhantes nas duas linguagens. É interessante observar que existem duas possíveis "ações" associadas à cláusula "else" tanto em CONIC quanto em Ada: uma refere-se à negação das expressões de guarda, ou seja, se todas as expressões de guarda retornarem um valor falso quando avaliadas, a cláusula "else" é satisfeita; a outra refere-se a um "delay 0" em Ada e "timeout 0" em CONIC, ou seja, se não houver pelo menos um processo ativo imediatamente pronto para o "rendezvous" com um "accept" (no caso de Ada) ou "receive" (em CONIC), a cláusula "else" também é satisfeita.

Não é natural esta última ação associada ao "else", pois já que tal condição pode ser expressa através da temporização de zero unidades de tempo, o "else" deveria funcionar apenas como a negação das expressões de guarda. Neste sentido, a sintaxe de CONIC é ainda mais obscura, pois permite a combinação das cláusulas "else" e "timeout" num mesmo "select", o que pode comprometer a clareza da construção "select" como um todo, devido à segunda função do "else". Em Ada, esta combinação não é permitida.

Além das duas formas principais de interação baseadas em passagem de mensagens, CONIC oferece a possibilidade de comunicação através de áreas compartilhadas. O compartilhamento é especificado através da definição de módulos segmentos [Sloman 85], onde apontadores para as áreas compartilhadas são passados de um processo a outro pelas mesmas primitivas utilizadas para passagem de mensagens, o que permite uma uniformidade na interação entre processos, independente da forma de comunicação adotada.

4. CONCLUSÃO

Apresentamos os principais conceitos e notações envolvidas em programação concorrente, onde enfatizamos as construções mais recentes de passagem de mensagens. Utilizamos linguagens como CSP, Ada e CONIC para ilustrar os principais conceitos envolvidos em passagem de mensagens. CSP foi utilizada para ilustrar uma solução simples e elegante de comunicação entre processos. Ada e CONIC são representativas de uma grande classe de linguagens de programação concorrente que reflete o "Estado da Arte" em programação distribuída.

Embora pertencentes à mesma classe, segundo a classificação proposta em [Andrew 83], CONIC oferece em vários aspectos uma maior flexibilidade e, muitas vezes, uma maior naturalidade do que Ada. Alguns destes aspectos incluem a possibilidade de comunicação onde a primitiva de envio de mensagens é assíncrona (transação "notify"); suporte direto de aplicações de vários clientes e vários servidores, o que é uma das inúmeras vantagens obtidas pela separação de programação e configuração; e mecanismos mais flexíveis para a deteção de falhas durante o "rendezvous", dentre outros.

Por outro lado, as primitivas de Ada são mais estruturadas (principalmente em se tratando das primitivas de chamada a procedimento remoto), evitando a programação de construções sem sentido -o que ocorre em CONIC- que podem comprometer a clareza dos programas e dificultar o desenvolvimento de técnicas de verificação de programas.

Como trabalhos em desenvolvimento, estamos estudando algumas restrições a serem impostas às construções de CONIC, a fim de fornecer

uma programação mais clara. A médio prazo, pretendemos desenvolver um mecanismo de verificação (semi-)automático para programação concorrente de linguagens do tipo Ada e CONIC, provavelmente, através de adaptações dos algorítimos apresentados em [Cunha 81, Taylor 83].

REFERENCIAS

[Andrew 83] Andrews G. R. and Schneider F. B. - "Concepts and Notations for Concurrent Programming"; Computing Surveys, ACM, Vol. 15, No. 1. March 1983.

[Campbe 76] Campbell R. H. - "Path Expressions: A Technique for Specifying Process Synchronization"; Ph.D. Dissertation. Computing Laboratory, University of Newcastle Upon Tyne, August 1976.

[Cunha 81] Cunha P. R. F. and Maibaum T. S. E. - "A Synchronized Calculus for Message Oriented Programming"; Proc. of 2nd International Conference on Distributed Computing Systems, April 1981.

[Dijkst 68] Dijkstra E. W. - "Cooperating Sequential Processes"; F. Genuys (Ed.), Programming Languages. Academic Press, New York, 1968.

[Dijkst 75] Dijkstra E. W. - "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs"; Commun. ACM, 18, 8 (Aug. 1975), 453-457.-

[Dulay 84] Dulay N., Kramer J., Magee J., Sloman M. and Twidle K. - "The CONIC Configuration Language"; Research Report, Dept. of Computing, Imperial College, August 1984.

[Hoare 72] Hoare C. A. R. - "Twards a Theory of Parallel Programming"; C. A. R. Hoare and R. H. Perrott (Eds.), Operating Systems Techniques. Academic Press, New York, 1972, pp 61-71.

[Hoare 74] Hoare C. A. R. - "Monitors: an Operating System Structuring Concept"; Commun. ACM 17,10 (Oct. 1974), 549-557.

[Hoare 78] Hoare C. A. R - "Communicating Sequential Processes"; Commun. ACM 21,8 (Aug. 1978). 666-677.

[Kramer 81] Kramer J., Magee J. and Sloman M. - "Intertask Communication Primitives for Distributed Computer Control Systems"; 2nd Int. Conf. Distributed Computing Systems, Paris, April 1981, 404-411.

[Kramer 84] Kramer J., Magee J., Sloman M., Twidle K. and Dulay N. - "The CONIC Programming Language: Version 2.4"; Research Report, Imperial College. Doc 84/19, Oct. 1984.

[Pyle 81] Pyle I. C. - "The Ada Programming Language"; Prentice Hall, 1981.

[Sloman 85] Sloman M., Kramer J. and Magee J. - "The CONIC Toolkit for Building Distributed Systems";6th IFAC Distributed Computer Control System Workshop, Monterey, California, USA, May 85.

[Taylor 83] Taylor R. N. - "A General Purpose Algorithm for Analyzing Concurrent Programs"; Commun. ACM, 26, 5 (May 1983).

[USDoD 81] U.S. Department of Defense - "Programming Language Ada: Reference Manual, Vol. 6"; Lecture Notes in Computer Science Springer-Verlag, New York, 1981.