Implementação de Delta-Prolog e Retrocesso Distribuído

José Cardoso e Cunha Joaquim Nunes Aparício Luis Moniz Pereira

Departamento de Informática Universidade Nova de Lisboa
2825 Monte da Caparica Setembro de 1985

Resumo:

O estado corrente da implementação da linguagem de programação em lógica Delta-Prolog é descrito Delta-Prolog é uma implementação parcial da **Lógica Distribuida** de Luis Monteiro que faz uma extensão ao Prolog com resolução paralela de goals e a noção de ordenação parcial de eventos

A unificação entre goals de eventos em dois processos baseia-se na proposta de Luis Moniz Pereira e usa mailboxes para comunicação entre processos

Delta-Prolog suporta programas distribuidos num processador único ou sobre uma rede, executando em paralelo múltiplas instâncias de um interpretador de C Prolog

A implementação actual sobre o VAX/VMS baseia-se no interpretador de C Prolog acrescido dos predicados necessários para a o controlo do paralelismo e comunicação entre processos a nivel da linguagem

Descrevem-se os niveis da implementação, incluindo os predicados de sistema e os algoritmos de controlo do retrocesso distribuido

Discutem-se os mecanismos básicos requeridos por Delta-Prolog nomeadamente controlo de processos, comunicação entre processos e retrocesso distribuido

Faz-se uma descrição pela primeira vez dos mecanismos para o suporte do retrocesso distribuido a interrupção de um processo Prolog e a facilidade do seu tratamento em Prolog

Apresentam-se exemplos de utilização de Delta-Prolog

Delta-Prolog

O trabalho de investigação em torno de Delta-Prolog envolve diversas dimensões fundamentais na área do processamento em Lógica Distribuida

- modelos de computação em Lógica Distribuida
- construções concorrentes ao nivel da linguagem
- suporte de desenvolvimento de ambientes de programação em lógica
- modelos de execução de programas em Lógica Distribuida
- ambientes de execução para o suporte de Lógica Distribuida
- aplicações em Inteligência Artificial

Delta-Prolog [3] é uma implementação parcial da Lógica Distribuida [1], que faz uma extensão ao Prolog com resolução paralela de golos, a noção de ordenação parcial de eventos.

A comunicação é binária e cada processo contribui com a sua parte complementar do evento (denotada pelos predicados ! e ?)

Dois processos em nós diferentes comunicando através de um evento ev devem ter os golos:

processo 1	processo 2
Termo_1 ! ev	Termo_2 ? ev

em que **ev** poderá eventualmente ser um nome da forma **no::ev** referenciando o nó **no**, e ambos resolvem se Termo_1 e Termo_2 unificam.

Na realidade pode-se usar um único operador (! ou ?), embora por razões de implementação se utilizem ambos. O mecanismo de comunicação por eventos é bidireccional, e simétrico. Essa simetria será no entanto implementada em futuras versões.

O estado corrente do Delta-Prolog é o resultado de um processo que evoluiu em três etapas

- o trabalho teórico de Luis Monteiro sobre Lógica Distribuída [1], introduzindo um modelo baseado em eventos para a comunicação entre processos, e a proposta de Luís Moniz Pereira [3] conduziram a um primeiro protótipo; a unificação entre goals de eventos entre dois processos foi implementada em Prolog por Luis Moniz Pereira e usa mailboxes na implementação sobre o VAX/VMS de cada comunicação; a descrição do mecanismo e sua concepção em Prolog é feita em [3].
- o trabalho desenvolvido por José C. Cunha e Joaquim N Aparício [2], neste Departamento melhorou a primeira versão e integrou-a num ambiente de rede de computadores, durante o segundo semestre de 84;
- a procura de soluções para o problema do retrocesso distribuído conduziu à introdução de novos mecanismos no ambiente de execução, de acordo com as propostas, de solução e implementação feitas por Luis M. Pereira.

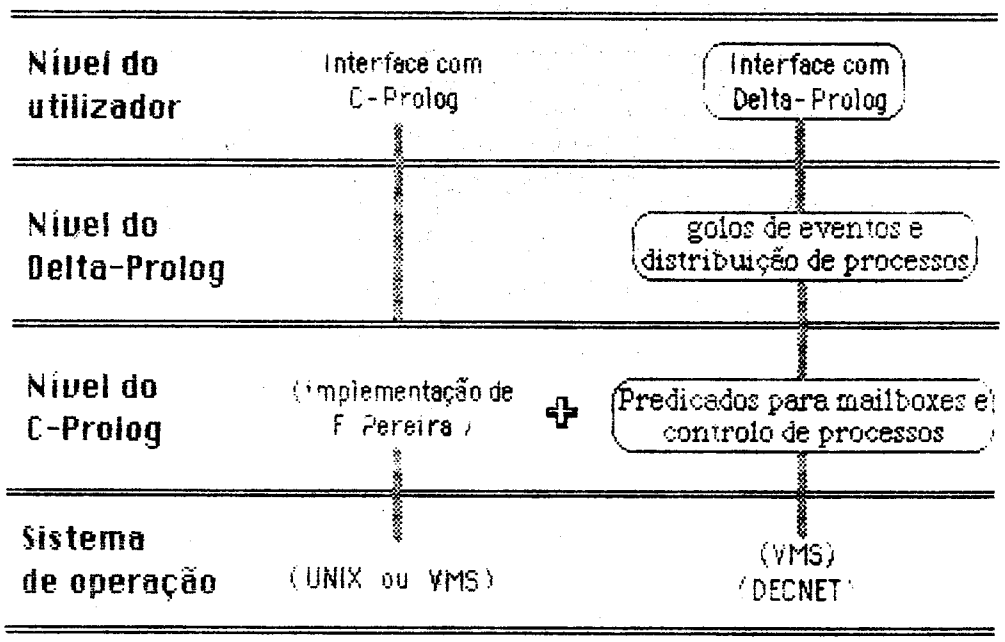
Algumas decisões influenciaram directamente o ambiente corrente de implementação:

- os protótipos têm a maioria do código escrito em Prolog, sendo assim fácil a sua adaptação a soluções alternativas permitindo ainda um crescimento gradual; este foi um dos objectivos principais, mesmo implicando uma menor eficiência da implementação
- os mecanismos de sistema requeridos por Delta-Prolog, nomeadamemte as facilidades para multiprocessamento e comunicação entre processos são suportados por uma camada que faz a interface com o ambiente de execução disponivel sobre a arquitectura de computadores, e encapsula os mecanismos não portáveis (prevendo-se a sua extensão para ambientes distribuídos heterogéneos);
- a implementação deve ser baseada, para facilitar o desenvolvimento, num Prolog existente (compilador ou interpretador), ao qual as extensões necessárias são feitas.

Ambiente Delta-Prolog

A versão corrente do Delta-Prolog é baseada no C-Prolog sobre o VAX/VMS, e assenta em software da Decnet quando existe comunicação entre nós de uma rede.

A implementação pode ser ilustrada do seguinte modo:



São suportados predicados de sistema para o controlo de processos e de mailboxes, bem como facilidades para execução de programas distribuídos:

- criação e destruição de processos;
- facilidades para associar canais de entrada/saída (nomeadamente mailboxes) aos processos criados;
- suporte de operação através da rede incluindo facilidades para o controlo remoto de processos e mailboxes e para a comunicação entre processos remotos;

Estas facilidades estão embebidas no código C como extensões ao interpretador de C-Prolog.

Estado actual. Paralelismo e Retrocesso Distribuído.

No estado actual é suportada a execução concorrente de processos que comunicam através de goals de eventos, ao nível da linguagem.

Ao nível mais primitivo, na implementação actual, um processo corresponde à execução do código do interpretador de C-Prolog, com as extensões acima referidas.

Uma das questões que se colocam, no ambito da investigação em torno de delta-Prolog, é a da integração do conceito de processo ao nível da linguagem. O estado actual de Delta-Prolog concretiza o conceito de processo pela execução de um interpretador, escrito em Delta-Prolog. Este interpretador recebe um goal na ocorrência de um evento, resolve-o e devolve uma solução usando outro evento:

```
proc <- G ? ev, G, G ! ev, proc .
```

sendo o pedido de execução de um goal:

```
pedido <- G ! ev, G ? ev
```

Execução de goals em paralelo

Admitindo assim que já existe um processo Delta-Prolog em execução, a resolução de dois goals em paralelo é

```
G1 // G2 <- G2 ! ev, G1, G2 ? ev
```

em que G2 é resolvido por um outro processo e G1 é resolvido localmente O operador // tem uma precedência superior à do operador , .

Retrocesso Distribuído

Um dos problemas que se coloca na resolução de goals em paralelo é garantir que o espaço de soluções da conjunção de G1 e G2 é completamente explorada. A implementação corrente garante ainda que a sequência das soluções obtidas é a mesma que a obtida por um interpretador sequencial de Prolog usando uma estratégia de depth-first e retrocesso para a procura de alternativas.

Assim se tivermos o conjunto de cláusulas

```
a(1) <-
a(2) <-
b(1) <-
b(2) <-
```

o conjunto de soluções para a query `<- a(X) // b(Y)` será

```
X=1 , Y=1
X=1 , Y=2
X=2 , Y=1
X=2 , Y=2
```

tal como em `<- a(X), b(Y)`

Estratégia de controlo

Considere-se a execução paralela dos goals A e B (A // B), sendo A executado por P1 e B executado por P2

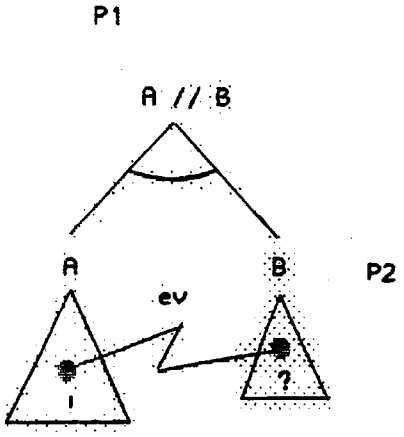


Figura 1

havendo comunicação através do evento ev

O problema de cooordenação surge quando, depois de a comunicação ter sido bem sucedida, um dos processos faz retrocesso até ao goal do evento: a sequência de

execução do outro processo deve ser alterada de modo a reflectir o retrocesso do primeiro.

A figura acima representa uma conjunção, pelo que, independentemente de qual dos processos retroceda até ao ponto do evento, deve ser sempre o processo P2 a explorar soluções alternativas (anteriores ao ponto de comunicação). O operador // estabelece uma relação de ordem entre goals (processos), que define a estratégia de controlo do retrocesso distribuido.

Evolução temporal dos processos no caso de P1 retroceder até ao evento

```
                P1                              P2

S1              G!ev                            S2   G?ev

                se
                retrocesso ate S1
                entao
                envia_comando(do_fail(S2))
                retry(S1)                       fail(S2)
```

Evolução temporal dos processos no caso de P2 retroceder até ao evento

```
                P1              P2

S1              G ! ev                          S2   G ? ev

                                                se
                                                retrocesso ate S2
                                                entao
                                                envia_comando(do_retry(S1))
                retry(S1)                       fail(S2)
```

em que :

envia_comando(X) usa o mecanismo de interrupções do Delta-Prolog, e é usado para enviar (de um modo assincrono) o termo X ao outro processo.

Quando X-do_retry(S), o processo interrompido executa retry(S), que é um predicado introduzido em Delta-Prolog que transfere o controlo do interpretador para o goal número X, desfazendo todas as unificações desde S até ao momento da execução.

Quando X-do_fail(S), o processo interrompido executa fail(S) que invoca retry(S), e faz falhar o goal de eventos número S.

Implementação

A ordem relativa entre os processos é mantida actualizando uma estrutura global acedida em exclusão no momento do lançamento em paralelo um novo goal (//) Essa estrutura é consultada quando um processo retrocede até um ponto de comunicação para determinação da ordem entre os dois processos envolvidos no evento em questão (predicados ! e ?).

Em cada evento (no caso de ser bem sucedido) os processos trocam entre si os números do goal referente ao evento. Posteriormente em retrocesso esse número é utilizado para controlar o parceiro envolvido na comunicação.

Exemplo (em que o processo P2 é um processo Delta-Prolog, que executa um goal recursivo p2, e P1 executa a query : <- a(X) // b(Y):

Cláusulas :

```
a(1) <- .
a(2) <- .
b(1) <- .
b(2) <- .
```

processo P1 : b(Y) ! ev, a(X), b(Y) ? ev.
processo P2 : p2 :- G ? ev, repeat, (G, G ! ev ; $fail ! ev), p2.
p2 :- fim.

NOTA : $fail é um termo que quando presente em um evento por qualquer dos processos provoca o falhanço do evento em ambos os processos.

A figura seguinte representa a ávore AND-OR a ser percorrida por cada um dos processos.

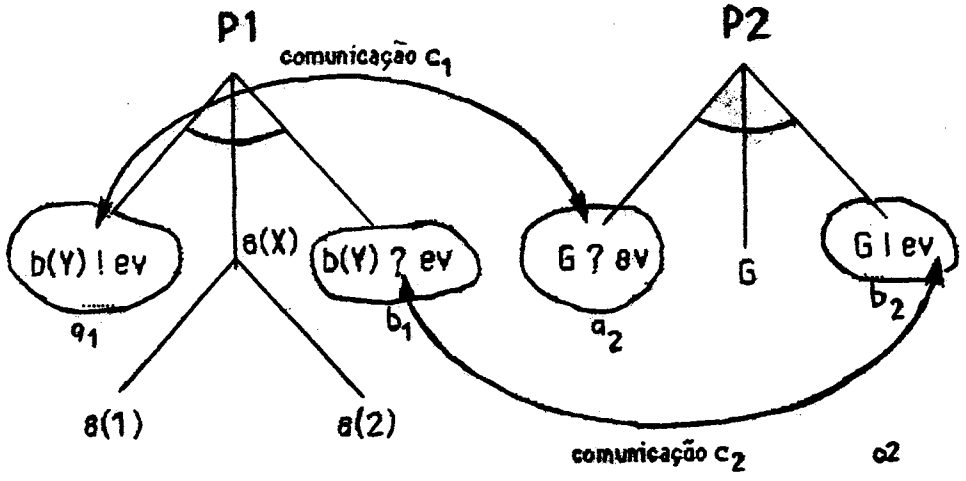


Figura 2

P2 encontra-se inicialmente em a_2 "esperando" um goal para resolver.

Quando P1 "envia" o goal b(Y) a comunicação c_1 resulta com a unificação de G com b(Y) em P2.

Seguidamente P1 resolve a(X) obtendo a(1), e paralelamente P2 resolve b(Y) obtendo b(1). Os processos sincronizam-se agora na comunicação c_2 em que P1 instancia b(Y) com b(1), resolvendo o goal inicial a(1) // b(1). Depois da comunicação c_2, P2 mantém-se à espera de novo goal para resolver em a'_2 (nova invocação de a_2 na chamada recursiva de P2).

Se P1 depois de ter obtido uma solução para o goal inicial, retroceder até à sua comunicação anterior (em b_1), envia um sinal a P2 que deve retroceder para o goal anterior ao da comunicação c_2, enquanto ele espera pela nova solução em b_1.

P2 retrocede para o goal b(Y) resolvendo agora com b(2) e executa a comunicação em b_2, resolvendo com Y instanciado com 2. P1 obtem assim a segunda solução com X=1, Y=2.

P2 depois de ter resolvido a segunda vez a comunicação espera novamente em a'_2.

Se P1 retroceder novamente para a comunicação em b_1, espera uma terceira solução de b(Y), que não existe e nesse caso a comunicação falha em ambos os processos. P1 retrocede até a(X) onde obtém a(2), e P2 retrocede até repeat,

reiniciando a procura de soluções para b(Y), devolvendo já o resultado. A execução repete-se agora com a(X) = a(2) para as diferentes soluções de b(Y)

A procura de soluções conjuntas termina quando P2 espera um novo goal para executar, em a'_2, e P1 retrocede ele mesmo até ao ponto da execução do pedido (em a_1), em que manda P2 falhar a recepção do pedido de b(X), "terminando" P2 e falhando P1 a query

Predicados de sistema

As extensões ao interpretador de C-Prolog suportam o ambiente de execução de Delta-Prolog

controlo de processos
- comunicação entre processos
- retrocesso distribuído

Uma camada de funções, implementadas em C, fornece o acesso aos serviços do VAX/VMS, ou do software da DECNET/VMS quando uma comunicação entre dois nós está envolvida.

Controlo de processos.

A distribuição de processos em Delta-Prolog, exige a capacidade de controlar processos concorrentes, no mesmo computador ou em diferentes nós da rede.

Os seguintes predicados de sistema foram implementados para permitir a criação suspensão, activação e destruição de processos locais ou remotos, a partir de um programa distribuido em Delta-Prolog.

Criação de processos

• **createp**(nomeprocesso,imagexec,input,output,mbxint)

cria um processo chamado nomeprocesso, executando o código em imagexec, (um ficheiro executável do VAX/VMS), com canais de I/O input e output, e uma mailbox mbxint para controlo das interrupções provocadas por outros processos.

Destruição de processos

• **killp**(nomeprocesso).

destroi o processo nomeprocesso.

Suspensão e reactivação de processos

• **suspnd**(nomeprocesso)

suspende um processo.

• **resumes**(nomeprocesso).

reactiva um processo préviamente suspenso.

Informação sobre processos

• **process_name**(Nome).

devolve o nome do processo.

O controlo de processos remotos é também possível através da simples designação dos nós envolvidos usando nomes compostos da forma "nó::nomeprocesso"

Comunicação entre processos.

Prolog fornece predicados de sistema para aceder a canais de entrada e saída (i/o), que são usados para ler e escrever de/para mailboxes de uma forma

transparente A comunicação remota é também suportada usando os mesmos predicados

Controlo de mailboxes em Delta-Prolog

Em Delta-Prolog há predicados de sistema para controlo explicito de mailboxes a partir de um programa:

- **cmbx**(nomelogico)
  cria uma mailbox nomelogico se não existir ainda, e abre sempre um canal para ela.

- **dmbx**(nomelogico)
  marca uma mailbox existente para destruição de tal modo que seja destruida quando não houver mais canais abertos para ela

- **contains_info**(nomelogico).
  tem sucesso se a mailbox nomelogico contém alguma informação senão falha

Comunicação numa rede de VAX's

A comunicação é também permitida entre dois processos Delta-Prolog em nos diferentes de uma rede de computadores

O sistema Delta-Prolog corre numa rede de computadores VAX-11/730 numa implementação que recorre ao sofware da Decnet-VMS

Para suportar o controlo de mailboxes e processos em nó remoto usando a DECNET, foi necessário definir objectos responsáveis pela implementação de pedidos recebidos de outro nó Estes objectos são escritos em C

Mecanismos primitivos para o suporte do Retrocesso Distribuido

Exige-se que o ambiente de execução forneça as seguintes facilidades
- mecanismo para interromper assincronamente um processo Prolog
- mecanismo primitivo para controlar o fluxo de execução do interpretador de Prolog (retry)

Predicados de sistema

- **send_interrupt**(process, term)
  Interrompe a execução de um processo Delta-Prolog e envia-lhe um termo

- **interruption** e **interruptoff**
  permitem e inibem o atendimento de interrupções de modo a garantir secções de código indivisiveis.

- **goalno**(Numero).
  devolve em Numero o número de invocação do goal corrente

- **retry**(Numero)
  retoma a execução a partir do golo de número de invocação Numero, fazendo retrocesso até àquele ponto.

A implementação do controlo de interrupções no VAX/VMS recorre a um mecanismode "Assynchronous System Traps". Qualquer interrupção destinada a um processo é detectada ao nivel do interpretador de C-Prolog, a partir do qual se

desencadeia a activação de um goal. Este goal é responsável pelo efectivo tratamento da interrupção (nomeadamente a recepção do termo enviado), totalmente programado em Prolog. Esta facilidade permite que o próprio utilizador defina em Prolog o comportamento do seu programa face 'as interrupções.

Predicados de Sistema em Delta-Prolog

- **spawn**(job,files).
  inicia a execução de um processo Delta-Prolog de nome "job" "files" é uma lista de ficheiros a ser consultado contendo um conjunto de cláusulas a serem usadas para a resolução de goals paralelos.

- **fork**(goal_1,goal_2) - "goal_1 **AND** goal_2"
  Lança a resolução do goal goal_2 em um processo Delta-Prolog. e resolve (em paralelo) o goal goal_1

Agradecimentos

Não pode passar sem referência a contribuição resultante das discussões com Luis Monteiro em torno do problema do retrocesso distribuido.

Referências

[1] Monteiro, Luis, A proposal para distributed programming in logic, em "Implementations of Prolog". (J. Campbell, ed.), Ellis Horwood 1984.

[2] Pereira, Fernando (ed.), C-Prolog User's Manual.

[3] Pereira, Luis Moniz; Nasr, Roger, Delta-Prolog a distributed logic programming language, em Proceedings of Fifth Generation Computer Systems, Tokyo, Nov. 84.

[4] Cunha, José C.; Aparicio, Joaquim N., UNL-6/84 (Dec 84) Departamento de Informática - Delta-Prolog Implementation : Progress Report Nº 1

[5] Cunha, José C.; Aparicio, Joaquim N., UNL-20/85 (Jul 85) Departamento de Informática - Delta-Prolog Implementation : Progress Report Nº 2