



Multi-Agent Programming Contest

[MAPC](#)

[Contest 2018](#)

[Publications](#)

[MASSim in Teaching](#)

[History](#)

The 2017 Contest

Ciclo baseado em steps

Para poder participar do jogo, o agente precisa:

- Esperar por um evento +step(X)
- Responder qual ação pretende executar. Exemplo: goto(shop4)

Impacto no estilo de programação:

- +step(X) provoca !intenção
- !intenção termina em ação

Ciclo baseado em steps

- +step(X)
- Descobrir o que estava fazendo antes
- Verificar se algo mudou
- Decidir o que fazer
- ação()

Exemplo

1a interação:

- +step(1)
- Agente ainda não tem um job
- Escolhe um job
- Escolhe uma loja
- goto(shop4)

Exemplo

2a interação:

- +step(2)
- Agente descobre o que estava fazendo antes
- Ainda não atingiu objetivo
- goto(shop4)

Exemplo

Algumas interações depois:

- +step(5)
- Agente descobre o que estava fazendo antes
- Finalmente chegou na loja
- Decide o que fazer em seguida
- buy(item1)

Consequências

- Lógica de programação concentrada nas condições de tratamento dos eventos +step(X): só um tratamento pode ser escolhido
- Condições exageradamente extensas
- Falta de comprometimento com um plano: as decisões têm curto prazo
- Alteração em um trecho de código pode afetar trechos não desejados
- Dificuldade de alterar a estratégia: necessário revisar todo o código
- Bugs são muito difíceis de corrigir

Solução proposta

Separação em 2 camadas, com diferentes tempos de resposta:

- Plan layer: definição da estratégia e da sequência de passos
- Step layer: responder ao eventos +step(X)

Solução proposta

Separação em 2 camadas, com diferentes tempos de resposta:

- Plan layer: definição da estratégia e da sequência de passos
 - Não precisa enviar ações ao servidor do jogo
 - Não precisa terminar em uma rodada
 - Define as intenções atuais
- Step layer: responder ao eventos +step(X)
 - Não toma decisões estratégicas
 - Descobre qual a intenção vigente e envia ações ao servidor do jogo
 - Informa quando as intenções foram atingidas ou falharam

Solução proposta

Exemplo

- Plan layer: tentar ir à loja shop4
 - A intenção fica ativa por várias rodadas
 - Sabe que a próxima intenção é comprar um item
- Step layer:
 - Descobre a intenção atual
 - Está na loja?
 - Sim: marcar intenção como atingida
 - Não: goto(shop4)

Código decepcionantemente simples que faz isso

```
+!try(goto(Y))  
<-  
  !step(X);  
  .wait(false);  
.  
  
+step(X) <- !step(X) .
```

```
+!step(X) // reached destination  
: .intend(try(goto(Y)))  
  & facility(Y)  
<-  
  .succeed_goal(try(goto(Y)));  
.  
+!step(X) // on route, but must recharge  
: .intend(try(goto(Y)))  
  & batteryOut  
<-  
  recharge;  
.  
+!step(X) // on route  
: .intend(try(goto(Y)))  
<-  
  goto(Y);  
.
```

Versão genérica para outras ações

```
+!try(Action)
<-
  Action;
  .wait(false);
.
```

```
+!step(X)
: .intend(try(Action))
& lastActionResult(successful)
<-
  .succeed_goal(try(Action));
.

+!step(X) //lastActionResult not successful
: .intend(try(Action))
<-
  .fail_goal(try(Action));
.
```

Uma outra abstração parecida: !retries

```
+!retries(M, Intention)
<-
    !!sub_retries(M, M, Intention);
    .wait(false);
.
```

```
+!sub_retries(M, N, Intention)
<-
    !Intention;
    .succeed_goal(retries(M, Intention));
.
-!sub_retries(M, N, Intention)
  : N > 1
<-
    !sub_retries(M, N-1, Intention);
.
-!sub_retries(M, N, Intention)
<-
    .fail_goal(retries(M, Intention));
.
```

Curiosidade

- Plan layer: definição da estratégia e da sequência de passos
 - Step layer: responder ao eventos +step(X)
-
- Plan layer: comportamento assíncrono em relação ao servidor
 - Step layer: comportamento síncrono em relação ao servidor

Logo a solução proposta é um modelo GSLA:

Globalmente Síncrono Localmente Assíncrono !!!!

Estratégia de teste: !solo

```
+!solo      // new job
: .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems;
  !buyItems;
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge;
  !try(goto(Storage));
  !retries(4, try(deliver_job(Job)));
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
!pick_job_solo; <=====
?doing(Job, Me);
!retrieveItems;
!buyItems;
?job(Job, Storage, Reward, Start, End, Items)
!maybe_charge;
!try(goto(Storage));
!retries(4, try(deliver_job(Job)));
!job_done(Job);
.print("done: ", job(Job, Reward))
!leave_job(Job);
!maybe_charge;
!step(_)
```

Coordena com outros agentes para
que ninguém pegue o seu trabalho

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems; <=====
  !buyItems;
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge;
  !try(goto(Storage));
  !retries(4, try(deliver_job(Job)));
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Itens de jobs que falharam podem ser guardados em storage. Verifica se há itens disponíveis. Evita desperdício.

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems;
  !buyItems;      <=====
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge;
  !try(goto(Storage));
  !retries(4, try(deliver_job(Job)));
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Escolhe a loja mais perto que tem um item necessário, vai até lá e compra o que precisar. Repete até que não precise comprar mais nada.

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems;
  !buyItems;
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge; <=====
  !try(goto(Storage));
  !retries(4, try(deliver_job(Job)));
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Se a bateria está baixa (<40%), vai até a chargeStation mais perto e recarrega.

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems;
  !buyItems;
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge;
  !try(goto(Storage));<=====
  !retries(4, try(deliver_job(Job)));
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Vai até o local de entrega do job.

Estratégia de teste: !solo

```
+!solo      // new job
:  .my_name(Me)
& not doing(_, Me)

<-
  !pick_job_solo;
  ?doing(Job, Me);
  !retrieveItems;
  !buyItems;
  ?job(Job, Storage, Reward, Start, End, Items)
  !maybe_charge;
  !try(goto(Storage));
  !retries(4, try(deliver_job(Job))); <=====
  !job_done(Job);
  .print("done: ", job(Job, Reward))
  !leave_job(Job);
  !maybe_charge;
  !step(_)
```

Tenta fazer a entrega do job
(lembrando: o servidor de jogo
gera falhas aleatórias)

Estratégia de teste: !solo

```
+!solo    // new job
: .my_name(Me)
& not doing(_, Me)

<-
    !pick_job_solo;
    ?doing(Job, Me);
    !retrieveItems;
    !buyItems;
    ?job(Job, Storage, Reward, Start, End, Items)
    !maybe_charge;
    !try(goto(Storage));
    !retries(4, try(deliver_job(Job)));
    !job_done(Job);
    .print("done: ", job(Job, Reward))
    !leave_job(Job);
    !maybe_charge;
    !step(_)
```

Se tudo certo, job done!

10 execuções

