USO DA LINGUAGEM RS EM ROBÓTICA

Gustavo Vasconcelos Arnold [*]
Giovani Rubert Librelotto [*,1]
Pedro Manuel Rangel Santos Henriques [*]
Jaime Francisco Cruz Fonseca [*]

[*] *Universidade do Minho, Braga, Portugal*

Abstract: Este artigo destina-se a apresentar a linguagem Reativa Síncrona RS como ferramenta para o desenvolvimento de software para robótica, tanto industrial como móvel, criando um mais alto nível de abstração no desenvolvimento de programas. A linguagem RS tem características que facilitam bastante o desenvolvimento e manutenção de programas deste tipo, visto que é uma linguagem reativa, simples, paralela e distribuída, e de tempo real, características estas que são importantes para o desenvolvimento deste tipo de programas.

Keywords: Robótica, Linguagens de Programação.

1. INTRODUÇÃO

Atualmente, a programação de robôs é realizada a baixo nível, em linguagens estruturadas imperativas ou até mesmo em código final. Assim sendo, esta tarefa torna-se demorada e dispendiosa, sendo, portanto, pouco lucrativa.

Pretende-se, com este trabalho, apresentar uma linguagem de mais alto nível, com um caráter declarativo, para o desenvolvimento de sistemas robóticos. Com esta solução, o desenvolvimento de programas torna-se facilitado, melhorando-se o tempo/esforço de programação, uma vez que a programação é feita em uma linguagem de nível mais alto.

A linguagem utilizada é a linguagem Reativa Síncrona (Toscani, 1993), que será referida como RS. Esta linguagem foi destinada, inicialmente, à programação de núcleos reativos, a parte central e mais difícil de um sistema reativo [2]. Tais núcleos são responsáveis por toda a lógica de um sistema reativo, manipulando os sinais de entrada, realizando as reações e gerando os sinais de saídas. A partir de um programa escrito em RS, é gerado um autômato finito correspondente, isto é, que descreve formalmente, através de estados e transições de estado com ações associadas, a semântica do sistema especificado por este programa.

Em (Arnold, 1998) e em (Piola, 1998) foi verificado que a linguagem RS se adequava muito bem para o desenvolvimento de sistemas robóticos, sejam eles industriais ou móveis, melhorarando bastante a tarefa de programação. Os sistemas robóticos possuem características bastante semelhantes as dos sistemas reativos.

A linguagem RS constitui uma notação adequada para representar o comportamento de núcleos reativos, e consequentemente de sistemas robóticos, pois um programa é uma especificação quase direta das transformações internas e das

[1] Bolsista CNPq - Brasil

[2] Sistema reativo é um tipo de sistema que interage fortemente com um ambiente, devendo ser capaz de responder a estímulos externos que chegam numa ordem desconhecida, continuamente. Exemplos: controladores de processos industriais, interfaces, vídeo-games, etc...

emissões de sinais que devem acontecer para cada estímulo possível.

Para atingir o objetivo, o artigo está organizado em duas grandes seções, além desta e da conclusão: na seção 2, introduz-se a linguagem RS, apresentando uma idéia geral da sua sintaxe e das características que a individualizam e a tornam adequada para o fim em vista; na seção 3 mostra-se a utilização da RS na programação de robôs, através da apresentação de dois estudos de caso concretos e completos, um aplica-se a um robô industrial e outro a um robô móvel.

2. LINGUAGEM RS

A linguagem RS pode ser vista como uma linguagem para a programação de núcleos reativos. Um programa RS é formado por um conjunto de declarações que especificam sinais e variáveis compartilhadas, e por um conjunto de regras de reação. As variáveis compartilhadas são especificadas através da declaração *var* e os sinais são declarados como *input*, *output* ou *signal*, consoante sejam de entrada, de saída ou internos, respectivamente. As regras de reação permitem descrever o comportamento de um sistema reativo. A regra realizará uma etapa do funcionamento do sistema somente quando a condição de disparo, desta mesma regra, for verdadeira.

2.1 A sintaxe

Pode-se dividir um programa RS em duas partes: a parte declarativa, onde se encontram todas as declarações dos sinais que poderão interagir com o sistema reativo; e a parte funcional, onde podemos encontrar as regras de reação que determinam o comportamento do sistema. Na figura (1), a parte declarativa de RS estende-se até a declaração *initially*. A partir deste ponto, especifica-se a parte funcional.

A estrutura de uma regra de reação é a seguinte:

```
s_ext#[s_int] ===> [ação]
```

Um sinal externo seguido de um sinal interno entre [e], separados pelo caracter #. Note que nem um dos sinais é obrigatório, mas pelo menos um deles deve estar presente. Logo após, encontra-se uma seta, a qual separa a condição de disparo e as ações da regra. Uma condição de disparo é verdadeira quando todos os sinais de disparo da mesma estão ligados.

As ações de uma regra de reação podem ser de vários tipos. Destacamos os dois principais. Os comandos *emit* e *up*. O comando *emit* contém um sinal de saída externo entre parênteses, o qual será emitido ao ambiente externo. O comando *up*

```
module M : [
  input : I,
  output : O,
  signal : S,
  var : V ,
  initially : C,
  R
].

Onde: I = [i1; i2; ...; im]; m-1,
  é a lista de sinais de entrada;
O = [o1; o2; ...; on]; n-0,
  é a lista de sinais de saída;
S = [s1; s2; ...; sp]; p-0,
  é a lista de sinais internos;
V = [v1; v2; ...; vq]; q-0,
  é a lista de variáveis;
C = [c1; c2; ...; ct]; t-1,
  é a lista de comandos de inicialização;
R = r1; r2; ...; ru; u-1,
  é a lista de regras de reação.
```

Fig. 1. Forma geral de um programa RS.

contém, entre parênteses, um sinal interno, o qual é ligado quando a regra em questão é executada.

As declarações devem aparecer obrigatoriamente na ordem mostrada na figura 1. A lista *input* não pode ser vazia, pois não faz sentido um programa reativo que não possa ser estimulado. Os sinais externos (*input* e *output*) e os sinais internos (*signal*) podem ser puros ou valorados; no segundo caso eles possuem campos capazes de armazenar valores arbitrários. Por convenção, os nomes dos campos iniciam com letra maiúscula. As variáveis do programa assumem valores numéricos apenas e seus nomes iniciam com letra minúscula. A declaração *initially* permite atribuir valores iniciais a variáveis e especificar sinais internos que devam estar ligados no início da execução (isto é, antes da primeira reação do programa).

2.2 Características da Linguagem RS

A Linguagem RS adota a *hipótese de sincronismo*, isto é, ela assume que toda e qualquer reação é executada em tempo zero. Portanto, os sinais de saída são síncronos com os sinais de entrada, e o tempo somente se passa durante a atividade do ambiente externo. Essa suposição simplifica a semântica da linguagem e permite que os programas RS sejam compilados para autômatos finitos.

Esta linguagem tem como características: caixas de regras, módulos e os sinais. As caixas de regras e os módulos permitem estruturar os programas. Os sinais, em RS são utilizados para comunicação com o exterior e para sincronização interna. Um programa RS trabalha com variáveis clássicas, que

```
P -> M+
M -> C+ | R+
C -> R+
R -> s_ext#[s_int] ===> [ação]
```

Fig. 2. Gramática de um programa RS.

são compartilhadas a nível de módulo. Os sinais podem ser divididos em três conjuntos:

- `Sinais de entrada`: são sinais de comunicação que o programa recebe do ambiente externo e são os únicos que podem desencadear reações. São especificados com a declaração *input*.
- `Sinais de saída`: são enviados ao ambiente externo para indicar os resultados das reações. São especificados com a declaração *output*.
- `Sinais internos`: são usados para sincronização e comunicação interna de processos. Ainda são subdivididos em:
  - `Temporários`: são declarados com *t_signal* e estão sempre desligados no início de uma reação; são usados para comunicação interna durante o desenvolvimento de uma reação e desligados automaticamente ao final desta.
  - `Permanentes`: permanecem no estado em que se encontram até que sejam explicitamente alterados (podem passar ligados de uma reação para outra). São declarados com a primitiva *p_signal*.

Um programa RS é composto por um conjunto de *módulos*, onde cada módulo é composto por um conjunto de *caixas de regras*, e cada caixa é composta por um conjunto de *regras de reação*, conforme demonstrado na figura (2). Apesar de ter estes três níveis hierárquicos, cada programa fonte RS pode ser composto por somente um simples conjunto de regras de reação. As caixas permitem agrupar regras, logicamente relacionadas, em unidades sintáticas próprias para fins de ativação e desativação (Librelotto, 2001).

A linguagem RS incorpora também construções para o tratamento de exceções. Situações de exceção são originadas por condições especiais que causam mudanças bruscas no estado do sistema reativo. Essas condições podem ser sinalizadas do exterior ou serem detectadas (e sinalizadas) internamente. Sempre que uma condição de exceção é sinalizada, o programa sofre uma mudança brusca de estado assim caracterizada:

- Os módulos envolvidos na situação de exceção sofrem um *reset*, isto é, para cada módulo, todos os sinais internos são desligados e todas as caixas de regras são desativadas.
- Para cada módulo envolvido, um novo estado é definido pela regra de exceção correspondente à condição que foi sinalizada.

Para representar as possíveis condições de exceção são utilizados os eventos de exceção. Um evento de exceção, tal como um sinal normal, pode ser puro ou ser constituído por campos capazes de conduzir informações arbitrárias. Os eventos são definidos na declaração *on_exception* do módulo. A cada evento corresponde uma regra, a qual define o tratamento da condição de exceção associada ao evento. Para sinalizar (levantar) uma condição de exceção durante uma reação, utiliza-se o comando *raise*, o qual especifica um evento declarado no bloco *on_exception*. Por exemplo, *raise(error(X;Y))* sinaliza a condição de exceção *error* e passa os parâmetros X e Y para o tratamento dessa excecão. A linguagem permite que os eventos de exceção sinalizados em um módulo se propaguem para os outros módulos (todos os mecanismos para esta propagação estão descritos em (Toscani, 1993)).

2.3 Compilação e Execução de RS

O compilador RS foi escrito em *Prolog*; ele traduz os programas fontes para um conjunto de tabelas que descrevem uma máquina de estados similar à *Máquina de Mealy* (Hopcroft and Hullman, 1979). Como o código objeto não é um ficheiro executável, o sistema necessita de um interpretador para a execução do autômato. Além do núcleo reativo de controle, uma aplicação reativa requer uma implementação de uma interface I/O (para receber os sinais de entrada e para informar os sinais de saída) e um conjunto de procedimentos, para manipular os dados da aplicação.

Como ocorre com Esterel (Berry and Gonthier, 1992), Lustre (Halbwachs, 1993) e outras linguagens síncronas, RS não é uma linguagem autosuficiente; a interface I/O e os componentes de manipulação de dados devem ser providos por uma linguagem hospedeira, ou por um ambiente de execução.

A execução de um programa RS é composta de uma sequência de passos, onde cada passo consiste de uma execução paralela de todas as regras que possuem a condição de disparo verdadeira. A primeira ação de um passo é uma ação implícita, que desliga todos os sinais contidos nas regras de disparo que foram disparadas. Como a execução de uma regra pode ligar sinais, isso origina uma sequência de novos passos, que somente termina quando o conjunto de sinais ligados não são o suficiente para disparar uma regra de reação. Nessas situações, o programa espera até que um novo sinal externo seja provido pelo ambiente, o que pode iniciar uma nova reação (uma sequência de passos) (Toscani, 1996).

```
module mouse:
[input :[click,tick],
 output:[single, double],
 signal:[awaitClick,awaitAny],
 var   :[count],
 initially: [up(awaitClick)],
 tick#[awaitClick]===>[up(awaitClick)],
 click#[awaitClick]===>[count:=2,
                        up(awaitAny)],
 tick#[awaitAny]===>case [
   count>0 ---> [count:=count1,
                 up(awaitAny)],
   else    ---> [emit(single),
                 up(awaitClick)] ],
 click#[awaitAny]===>[emit(double),
                      up(awaitClick)]
].
```

Fig. 3. Programa RS para o rato.

```
AUTOMATON:
  init   -   [1,*,go_to(1)]
   1   tick  [2,*,go_to(1)]
   1   click [3,*,go_to(2)]
   2   tick  [[4 - 1,*,go_to(2)],
              [4 - 2,*,go_to(1)] ]
   2   click [5,*,go_to(1)]
```

Fig. 4. Autômato gerado para o rato.

```
RULES:
  Module mouse:
  1. [ ] ===> [ ]
  2. [ ] ===> [ ]
  3. [ ] ===> [count:=2]
  4. Case:
  4-1. [ ]{count>0} ---> [count:=count1]
  4-2. [ ]{else} ---> [emit(single)]
  5. [ ] ===> [emit(double)]
```

Fig. 5. Regras para o autômato da figura 4.

2.4 Exemplo

O exemplo da figura (3) é um controlador para um *rato* de uma única tecla. Os sinais de entrada são: *click* (carga da tecla) e *tick* (sinal de um relógio). O programa emite o sinal *double* quando recebe dois clicks simultâneos e o sinal *single* quando recebe um único click. Dois clicks são considerados simultâneos quando ocorrem separados por menos de 3 unidades de tempo.

Inicialmente é ligado o sinal *awaitClick*. Logo, podem ser disparadas apenas as duas primeiras regras do programa. Esta situação permanece até o programa ser estimulado por um *click*, que faz o contador igual a 2 e liga o sinal *awaitAny*. A partir daí, três *ticks* consecutivos fazem emitir o sinal *single* e um *click* faz emitir o sinal *double*. Depois da emissão de um sinal, o programa volta ao estado no qual só as duas primeiras regras podem disparar.

O autômato RS gerado pelo compilador é apresentado em duas partes: o autômato propriamente dito e as ações a serem executadas por este autômato. Na representação do autômato (figura (4)), cada linha tem 3 componentes: um número de estado n, um sinal de entrada s e uma lista (árvore) da qual se obtém a seqüência de ações a executar quando, no estado n, ocorrer o sinal s. Na lista, as ações são indicadas por números que identificam regras de execução, as quais são mostradas logo após o autômato (as ações estão no lado direito dessas regras). Os asteriscos separam os trechos correspondentes aos passos de execuções referidos anteriormente e toda seqüência termina com *go_to(k)*, onde k é o próximo estado do autômato.

No estado *init* são executadas as ações de inicialização do programa (a declaração *initially* origina uma regra de execução extra). Estas ações iniciais são executadas antes do autômato ser colocado em funcionamento (portanto, o estado inicial real do autômato é o estado 1).

As regras de execução (encontradas no arquivo de regras, descrito na figura (5)) são uma simplificação das regras do programa. Nelas desaparecem as referências aos sinais puros (que não conduzem informação). As informações de sincronização dos sinais são usadas em tempo de compilação, para sequencializar as ações do programa, e não são mais necessárias em tempo de execução. Convém observar que, neste exemplo, as regras 1 e 2 não têm ações para executar; elas poderiam ser eliminadas, desde que fossem eliminadas as referências a elas, no autômato.

Nas regras condicionais, as várias opções aparecem de forma explícita. Para cada opção é listado o lado esquerdo da regra, seguido da condição de execução entre chaves, seguido (após a seta simples) da lista de ações. A transição mais complexa do autômato acontece quando, estando no estado 2, ele recebe o sinal *tick*. Neste caso: se a primeira condição da regra 4 é verdadeira, então executa as ações da regra 4–1 e continua no estado 2; caso contrário, se a segunda condição da regra 4 é verdadeira, então executa as ações da regra 4–2 e vai para o estado 1.

3. PROGRAMAÇÃO DE ROBÔS EM RS

Para utilizar a linguagem RS no desenvolvimento de sistemas robóticos, é preciso:

- identificar quais dos sensores existentes no robô serão necessários para realizar determinada tarefa e definí-los como sinais internos do programa RS;
- especificar um sinal de entrada especial para dar início a execução do autômato, e con-

sequentemente do programa a ser executado pelo robô. Este sinal corresponde ao estado pronto do robô, que corresponde ao estado em que o mesmo se encontra ligado e com o programa a ser executado em sua memória;
- identificar que funções deverão ser executadas pelo robô para realizar determinada tarefa corretamente e definí-las como sinais de saída, que serão enviados para o ambiente externo através do comando *emit*. Os sinais de saída poderão conter parâmetros que sejam necessários para executar determinada função.

Se for necessário especificar pontos por onde o robô deverá passar, o programador RS não precisa se preocupar com os mesmos, apenas assume que eles já existem, preocupando-se apenas com a lógica do sistema. Os pontos podem ser gravados antes ou depois de feito um programa RS utilizando-se, por exemplo, de um *teach box*.

3.1 Exemplo de programa RS para controlar um robô industrial

Um exemplo de programa construído em RS e utilizado, na prática, em um robô Nachi SC15F pode ser visto na figura (6) (Piola, 1998).

Este exemplo apenas controla o robô de forma que o mesmo execute dez ciclos passando por três pontos, *P1, P2* e *P3*. No momento em que o robô se dirige para o ponto *P2*, o mesmo sofrerá um deslocamento determinado pela distância de *X, Y, Z*. Após passar por *P2* e enquanto se dirige para *P3*, o mesmo sofrerá novamente um deslocamento de forma a retornar as coordenadas iniciais.

Se, durante a execução deste programa, o sensor *sensor* for acionado, o robô assumirá outro comportamento, durante apenas um ciclo, devendo passar pelos pontos *P4, P5* e *P6*.

Quando compilado, este programa RS irá gerar um autômato, cujas caracteríísticas já foram descritas na seção 2.4. Em (Piola, 1998) foi desenvolvido um tradutor capaz de, a partir do autômato gerado pelo compilador de RS criar o programa a ser executado no robô Nachi SC15F. O programa gerado por este tradutor pode ser visto na figura (7).

Este tradutor tornou possível a utilização da linguagem RS no desenvolvimento de programas para o robô Nachi SC15F. Com isto, tornou o desenvolvimento de programas mais simples e rápido, visto que as linguagens dos robôs são específicas para os mesmos, tornando sua programação imperativa e dependente do mesmo, sendo desta forma, de um nível mais baixo. Através da linguagem RS foi possível abstrair-

```
rs_prog nr_0001:
[input: [inicio, sensor],
 output: [ir(P), fim, ferramenta(C),
          usar(B), deslocar(D,X,Y,Z)],
 module mov:
 [ input: [inicio, sensor],
  output: [ir(P), fim, ferramenta(C),
           usar(B), deslocar(D,X,Y,Z)],
  t_signal: [],
  p_signal: [t1, t2, t3, t4, t5],
  var: [count1],
  initially: [activate(rules),
              emit(usar(1)),
              emit(ferramenta(0)),
              count1:=0, up(t1)],
  inicio#[t1]===>case
                  [count1=10--->[count1:=0,
                         emit(fim),up(t1)],
             else--->[count1:=count1+1,
                    emit(ir(p1)),up(t2)],
      [t2]===>[emit(ir(p2)),
               emit(deslocar(0,0,0,100)),
               up(t3)],
      [t3]===>[emit(ir(p3)),
               emit(deslocar(0,0,0,0)),
               up(t1)],
  sensor#[t1]===>[emit(ir(p4)),up(t4)],
         [t4]===>[emit(ir(p5)),up(t5)],
         [t5]===>[emit(ir(p6)),up(t1)],
 ]
].
```

Fig. 6. Programa RS para controlar o robô.

```
10 USE 1
20 TOOL 0
30 V1%=0
40 *S1
41 JMPI 4, I1
50 IF V1%=10 THEN *L2X1 ELSE *L2X2
60 *L2X1
70 V1%=0
80 END
90 GOTO *S1
100 *L2X2
110 V1%=V1%+1
120 MOVE P,P1,T=3
130 MOVE P,P2,T=3
140 SHIFTA 0,0,0,100
150 MOVE P,P3,T=3
160 SHIFTA 0,0,0,0
170 GOTO *S1
180 MOVE P,P4,T=3
190 MOVE P,P5,T=3
200 MOVE P,P6,T=3
210 GOTO *S1
```

Fig. 7. Programa gerado para o robô.

se das características do robô, tornando sua programação declarativa e, desta forma, de mais alto nível.

3.2 Exemplo de programa RS para controlar um robô móvel

Outro exemplo de programa construído em RS é para simular o controle de um veículo móvel, chamado Tó (Corrêa, 1992), o qual deve vaguear, evitando obstáculos. Caso ocorra o aparecimento de algum objeto móvel de interesse, o Tó deve seguí-lo. Quando o objeto sai do seu alcance, o Tó volta novamente a vaguear. O veículo é composto de 3 sensores de raios infravermelhos, sendo 2 frontais e um traseiro, e dois motores. Os movimentos possíveis para os motores são: esquerda, direita e reto, para o motor de direção, e frente, trás e parado, para o motor de movimento.

Para implementar este programa, foi necessário fazer uma extensão na linguagem RS de forma que fosse possível decompor este sistema segundo uma modelagem baseada em comportamentos, no caso a Arquitetura de Subsunção (Brooks, 1986). Com esta extensão, tornou-se possível implementar qualquer sistema em RS segundo esta forma de modelagem. Para isto, foram adicionados na linguagem RS os mecanismos de controle: inibidores e supressores (Arnold, 1998).

Os inibidores possuem a sintaxe descrita abaixo:

```
<sinal1> inhibit <sinal2> for <tempo>.
```

onde *sinal1* corresponde ao sinal dominante; *sinal2* corresponde ao sinal inibido; e *tempo* corresponde ao período de tempo em que o inibidor permanece ativo. Desta forma, enquanto o sinal dominante estiver presente, o módulo em questão não emitirá o sinal inibido.

Os supressores possuem a sintaxe semelhante a dos inibidores:

```
<sinal1> supress <sinal2> for <tempo>.
```

onde *sinal1* corresponde ao sinal dominante; *sinal2* ao sinal suprimido; e *tempo* ao período de tempo em que o supressor permanece ativo. Desta forma, enquanto o sinal dominante estiver presente, este será o sinal emitido pelo módulo. Se não houver sinal dominante, o outro sinal poderá ser emitido.

Aplicando a modelagem baseada na Arquitetura de Subsunção no desenvolvimento do Tó, o programa é decomposto em módulos conforme mostrado na figura (8), sendo necessário, posteriormente, desenvolver cada um dos módulos separadamente. Somente após terminados todos os testes individuais, os módulos seriam unidos

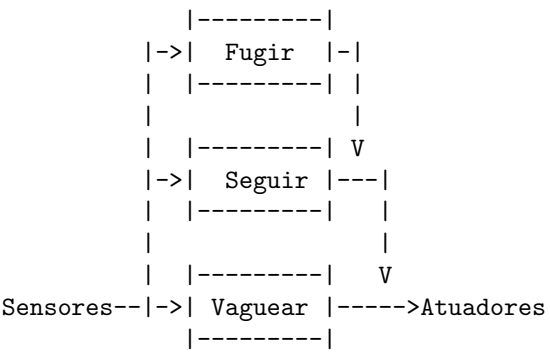

Fig. 8. Decomposição do Tó baseada na Arquitetura de Subsunção.

```
module vaguear:
[input :  [tick],
 output:  [v_andar(X), v_virar(Y)],
 t_signal: [next1(N)], p_signal: [],
 var : [],
 initially: [random_init(7),
             activate(rules)],
 on_exception: [],
 tick ===> [random(N), up(next1(N))],
 #[next1(N)]  ===> case
   [N <= 25 --->[emit(v_andar(frente)),
                 emit(v_virar(esq))],
    N <= 50 --->[emit(v_andar(frente)),
                 emit(v_virar(dir))],
    else    --->[emit(v_andar(frente)),
                 emit(v_virar(desl))] ]
].
```

Fig. 9. Módulo RS responsável pelo comportamento vaguear.

utilizando-se dos mecanismos inibidores e/ou supressores, compondo, desta forma, o sistema final.

Em uma versão anterior, apresentada em (Arnold and Toscani, 1998), os mecanismos de controle eram implementados pelo próprio programador, durante o desenvolvimento do sistema. Com esta nova extensão, o programador não precisa se preocupar, em tempo de desenvolvimento, com estes mecanismos. O mesmo deve desenvolver cada módulo separadamente e, quando terminar, começa-se a fase de ligação destes módulos, através destes mecanismos. Isto torna os módulos independentes uns dos outros (caixas pretas), sendo cada um responsável apenas por seu comportamento específico e com sua respectiva emissão de sinais, o que facilita a construção de sistemas mais complexos.

O módulo apresentado na figura (9), responsável por fazer o Tó caminhar aleatoriamente, recebe como entrada apenas o sinal de clock *tick*, e emite dois sinais de saída, *v_andar* e *v_virar*, que serão emitidos para o ambiente externo. Quando o sinal *tick* for recebido, será ativado um sinal temporário *next1*, que irá conduzir um número inteiro aleatório *N*. Este sinal temporário é responsável

```
module seguir:
[input :  [sIV(E, D, T, S, D1, D2)],
 output:  [s_andar(X), s_virar(Y)],
 t_signal: [], p_signal: [], var: [],
 initially: [activate(rules)],
 on_exception: [],
 sIV(E, D, T, S, D1, D2) ===> case
  [S=1 & D1=1--->[emit(s_andar(frente)),
                  emit(s_virar(desl))],
   S=1 & D2=0--->[emit(s_andar(frente)),
                  emit(s_virar(esq))],
   S=1 & D2=1--->[emit(s_andar(frente)),
                  emit(s_virar(dir))],
   else      ---> [] ]
].
```

Fig. 10. Módulo RS responsável pelo comportamento seguir.

```
module fugir:
[input :[sIV(E, D, T, S, D1, D2)],
output:[f_andar(X), f_virar(Y)],
t_signal: [], p_signal: [], var: [],
initially: [activate(rules)],
on_exception: [],
sIV(E,D,T,S,D1,D2) ===> case
[E=1& D=0& T=0--->[emit(f_andar(tras)),
                   emit(f_virar(esq))],
 E=0& D=1& T=0--->[emit(f_andar(tras)),
                   emit(f_virar(dir))],
 E=1& D=1& T=0--->[emit(f_andar(tras)),
                   emit(f_virar(desl))],
 E=0& D=0& T=1--->[emit(f_andar(frente)),
                   emit(f_virar(desl))],
 else         ---> [] ]
].
```

Fig. 11. Módulo RS responsável pelo comportamento fugir.

```
environment:-subsumption.
f_andar supress s_andar for 5.
f_virar supress s_virar for 5.
s_andar supress v_andar for 3.
s_virar supress v_virar for 3.
```

Fig. 12. Módulo RS responsável pelo comportamento fugir.

por disparar a regra que controla a direção e o sentido a ser tomado pelo veículo, sendo emitido os sinais de saída. Este módulo contém somente a programação do seu próprio comportamento, não precisando ativar nenhum mecanismo de controle, pois este é o comportamento básico do veículo. As saídas deste módulo poderão vir a ser suprimidas por algum outro módulo, mas isto fica sob responsabilidade do ambiente de execução.

O módulo apresentado na figura (11) é responsável pelo comportamento de fuga do veículo móvel, fazendo com que o mesmo evite obstáculos. Este módulo recebe como entrada apenas o sinal externo *sIV*, porém emite dois sinais de saída diretamente para a camada de interface com o ambiente externo, *f_andar* e *f_virar*, responsáveis pelo acionamento dos motores de movimento e de direção, respectivamente. Desta maneira, quando algum dos sensores *E, D* ou *T* estiver ligado, serão emitidos comandos para evitar os obstáculos, juntamente com comandos para suprimir os sinais que possam vir a ser emitidos pelo módulo seguir, por um determinado tempo, no caso 5 unidades de tempo. O ambiente de execução será responsável pelo correto funcionamento dos mecanismos de controle.

No módulo apresentado na figura (10), responsável por fazer o Tó perseguir algum objeto de interesse, recebe como entrada apenas o sinal externo *sIV*, como na versão anterior. No entanto, emite dois sinais de saída, *s_andar* e *s_virar*. Quando o parâmetro *S*, que indica a intensidade dos sensores dianteiros, valer 1, será disparada a regra que decide a direção e o sentido a ser tomado pelo Tó para perseguir o objeto. A perseguição é dirigida pelos valores dos parâmetros *D1* e *D2*. Além dos comandos para perseguir o objeto, são emitidos comandos para suprimir os sinais que possam vir a ser emitidos pelo módulo vaguear, por um determinado tempo, no caso 3.

A figura (12) contém as declarações para o ambiente de execução e devem ser incluídas no final do programa. Esta parte especifica as ligações entre os diversos módulos através dos mecanismos de controle. Inicialmente é declarado o tipo de ambiente de execução que será utilizado. O tipo de ambiente usado nesta extensão é chamado de *subsumption*. Este ambiente implementa os mecanismos de controle e deve ser declarado sempre que for utilizada esta extensão. Nas linhas seguintes, são declarados os supressores, indicando o sinal que irá suprimir, o sinal que será suprimido, e o intervalo de tempo em que cada supressor permanecerá ativo. Neste exemplo, o sinal *f_andar* irá suprimir o sinal *s_andar* por um período de tempo 5; o sinal *f_virar* irá suprimir o sinal *s_virar* por um período de tempo 5; o sinal *s_andar* irá suprimir o sinal *v_andar* por um período de tempo 3; o sinal *s_virar* irá suprimir o sinal *v_virar* por um período de tempo 3.

Desenvolvendo sistemas para veículos móveis desta forma, as principais características da Arquitetura de Subsunção que são a modularidade, simplicidade, independência e robustez, são incorporadas ao sistema, uma vez que cada módulo é responsável apenas por um comportamento, o que o faz com eficiência e segurança.

4. CONCLUSÃO

Este artigo teve como objetivo apresentar a linguagem RS aplicada à programação de sistemas robóticos, sejam eles móveis ou industriais. O uso desta linguagem apresenta uma série de vantagens, que podem ser resumidas em uma única frase: a criação de um nível mais alto de abstração para a programação de robôs.

Este nível de abstração mais alto torna o desenvolvimento de sistemas facilitado, mais rápido, seguro, modular, podendo até mesmo tornar os programas portáveis entre robôs diferentes, desde que os mesmos utilizem, como entrada, o autômato gerado pela linguagem RS.

Um estudo mais aprofundado está sendo realizado a respeito das características requeridas de uma linguagem para fins robóticos, principalmente industriais, a fim de realizar uma extensão da linguagem RS para torná-la uma linguagem geral para o desenvolvimento de sistemas deste tipo, tornando a tarefa de programação mais simples e escondendo do usuário detalhes específicos do robô. Isto facilitaria bastante a tarefa de programação dos robôs industriais, pois haveria um aumento da portabilidade, da reusabilidade e, desta forma, do tempo de vida dos programas, já que os mesmos poderiam permanecer válidos quando ocorresse a troca de equipamentos.

REFERENCES

Arnold, G. V. (1998). *Uma Extensão da Linguagem RS para o Desenvolvimento de Sistemas Reativos Baseados na Arquitetura de Subsunção*. Dissertação de Mestrado, CPGCC, UFRGS. Porto Alegre, Brasil.

Arnold, G. V. and S. S. Toscani (1998). Using the RS language to control autonomous vehicles. In: *Workshop on Intelligent Components for Vehicles*. pp. 465–470. International Federation of Automatic Control. Seville, Spain.

Berry, G. and G. Gonthier (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* **19**, 87–152.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of robotics and Automation* **RA-2, n. 1**, 14–23.

Corrêa, L. (1992). O veículo de três olhos (tó para os amigos). *Publicação interna. Lisboa: Departamento de Robótica, Universidade Nova de Lisboa.*

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Klumer Academic Publisher. Dordrecht.

Hopcroft, J. E. and J. D. Hullman (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley. Massachussetts, USA.

Librelotto, G. R. (2001). *Um Compilador para a Linguagem RS Distribuída*. Dissertação de Mestrado, PPGC, UFRGS. Porto Alegre, Brasil.

Piola, S. J. (1998). *Uso da Linguagem RS no Controle do Robô Nachi SC15F*. Trabalho de Conclusão de Curso de Graduação, Departamento de Informática, UCS. Caxias do Sul, Brasil.

Toscani, S. S. (1993). *RS: Uma Linguagem para Programação de Núcleos Reactivos*. Tese de doutoramento, Depto de Informática, UNL. Lisboa, Portugal.

Toscani, S. S. (1996). *Introdução aos Sistemas Reativos*. CPGCC, UFRGS. Porto Alegre, Brasil.