Uso da linguagem RS em Robótica

Gustavo V. Arnold, Giovani R. Librelotto[Ψ], Pedro R. Henriques, Jaime C. Fonseca
{gva, grl, prh}@di.uminho.pt, jaime.fonseca@dei.uminho.pt
Universidade do Minho, Braga, Portugal
[Ψ] Bolsista CNPq - Brasil

Resumo **- Este artigo tem como objectivo apresentar a linguagem Reactiva Síncrona RS como ferramenta de desenvolvimento de software para robótica, tanto industrial como móvel, criando um nível mais alto de abstracção 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 reactiva, simples, paralela e distribuída, características estas que são importantes para o desenvolvimento deste tipo de programas.**

Abstract **–This paper presents a Reactive and Synchronous language, called RS, as a tool to develop programs for robots, industrial or mobile, creating a higher abstraction level during the programming task. The RS language is simple to use and has constructions to express parallelism and distribution; those are important features that facilitate the development and maintenance of this kind of programs.**

I. INTRODUÇÃO

Actualmente, a programação de robôs é realizada a baixo nível, em linguagens estruturadas imperativas ou até mesmo em código final (assembly). 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 carácter declarativo, para o desenvolvimento de sistemas robóticos. Com esta solução, o desenvolvimento de programas torna-se facilitado, melhorando-se o tempo/esforço de programação.

Por linguagem de alto nível entende-se, neste caso, uma linguagem muito mais próxima da especificação da lógica que traduz o comportamento do sistema e que nos permite abstrair dos detalhes físicos concretos; nomeadamente, sendo uma linguagem declarativa, podemos concentrar-nos na reacção a executar por cada estímulo, sem nos preocuparmos com a ordem das instruções, como costuma acontecer nas linguagens imperativas. Subindo o nível de abstracção da linguagem, a programação torna-se mais fácil porque o programa (especificação) é mais curto (os comandos são mais complexos e expressivos e, portanto, em menor número) e mais próximo da nossa forma de raciocinar.

Segundo Groover [7], um robô pode ser programado através de um destes quatro métodos: *setup* manual; programação *leadthrough*; programação directa em linguagem de programação própria; e programação *off-line*. A linguagem utilizada neste trabalho permite uma programação off-line. No entanto, trata-se apenas de uma linguagem de programação de robôs, e não de um ambiente integrado de programação de sistems robóticos como é o caso do Workspace (www.workspace5.com).

As linguagens de programação de robôs habituais, como por exemplo o RAPID, o VAL, e a linguagem do robô NACHI SC15F, são linguagens imperativas e específicas de uma determinada família de robôs. Embora sejam de mais alto nível do que o código de máquina, ou o assembly, obrigam o programador a ter em mente o modelo de funcionamento do robô, as suas características específicas e a seguir o paradigma de programação procedimental, onde a ordem das instruções é relevante e o cuidado com o fluxo de controlo de execução é fundamental. Os programas desenvolvidos nestas linguagens não são portáveis, até mesmo dentro da mesma família. Logo, além de serem linguagens de mais baixo nível em comparação com as linguagens declarativas (os comandos são muito próximos das instruções de máquina), não facilitam a portabilidade nem a reutilização do código fonte.

A linguagem utilizada é denominada como Linguagem Reactiva Síncrona RS [12]. Esta linguagem foi destinada, inicialmente, à programação de núcleos reactivos, a parte central e mais difícil de um sistema reactivo[1]. Tais núcleos são responsáveis por toda a lógica de um sistema reactivo, manipulando os sinais de entrada, realizando as reacções e gerando os sinais de saída. A partir de um programa escrito em RS, é gerado um autómato finito correspondente, isto é, que descreve formalmente, através

[1] Sistema reactivo 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...

de estados e transições de estado com acções associadas, a semântica do sistema especificado por este programa.

Em [4] e em [10] foi verificado que a linguagem RS se adequava muito bem para o desenvolvimento de sistemas robóticos, sejam eles industriais ou móveis, melhorando bastante a tarefa de programação. Os sistemas robóticos possuem características bastante semelhantes às dos sistemas reactivos, daí justificariam esta adequação.

A linguagem RS constitui uma notação adequada para representar o comportamento de núcleos reactivos, e consequentemente de sistemas robóticos, pois um programa é uma especificação quase directa das transformações internas e das emissões de sinais que devem acontecer para cada estímulo possível.

Para atingir o objectivo, o artigo está organizado em duas grandes secções, além desta e da conclusão: na secção 2, introduz-se a linguagem RS, apresentando uma ideia geral da sua sintaxe e das características que a individualizam e a tornam adequada para o fim em vista; na secção 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.

II. LINGUAGEM RS

A linguagem RS pode ser vista como uma linguagem para a programação de núcleos reactivos. Um programa RS é formado por um conjunto de declarações, que especificam sinais e variáveis partilhadas, e por um conjunto de regras de reacção. As regras de reacção permitem descrever o comportamento de um sistema reactivo. A regra realizará uma etapa do funcionamento do sistema somente quando a condição de disparo, desta mesma regra, for verdadeira.

A. A sintaxe

Pode-se dividir um programa RS em duas partes: a parte das declarações, onde se encontram identificados todos os sinais que poderão interagir com o sistema reactivo; e a parte funcional, onde podemos encontrar as regras de reacção que determinam o comportamento do sistema. Na figura 1, a parte das declarações estende-se até a declaração *initially*. A partir deste ponto, especifica-se a parte funcional.

A estrutura de uma regra de reacção é a seguinte:

```
s_ext#[s_int] ===> [acção]
```

O lado esquerdo da regra é formado por um sinal externo seguido de um sinal interno entre [e], separados pelo caracter #. Note que nenhum 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 das acções (lado direito da regra). Uma condição de disparo é verdadeira quando todos os sinais de disparo da mesma estão activos (ligados).

As acções de uma regra de reacção podem ser de vários tipos. Destacamos os dois principais. Os comandos *emit* e *up*. O comando *emit* contém um sinal de saída como parâmetro, o qual será emitido para o ambiente externo. O comando *up* contém, como parâmetro, um sinal interno, o qual é ligado quando a regra em questão é executada.

```
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 reacção.
```

Fig. 1 - Forma geral de um programa RS.

As declarações devem aparecer obrigatoriamente na ordem mostrada na figura 2. A lista *input* não pode ser vazia, pois não faz sentido um programa reactivo 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 reacção do programa).

B. Características da Linguagem RS

A Linguagem RS adopta a *hipótese de sincronismo*, isto é, ela assume que toda e qualquer reacção é executada com dispêndio de tempo zero. Portanto, os sinais de saída são síncronos com os sinais de entrada, e o tempo somente se passa durante a actividade do ambiente externo. Essa suposição simplifica a semântica da linguagem e permite

que os programas RS sejam compilados para autómatos 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 são partilhadas 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 reacções. São especificados com a declaração *input*;
- Sinais de saída: são enviados ao ambiente externo para indicar os resultados das reacções. 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 reacção; são usados para comunicação interna durante o desenvolvimento de uma reacção e desligados automaticamente ao final desta;
  - Permanentes: permanecem no estado em que se encontram até que sejam explicitamente alterados (podem passar ligados de uma reacção 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 reacção*, conforme se vê 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 reacção. As caixas permitem agrupar regras, logicamente relacionadas, em unidades sintácticas próprias para fins de activação e desactivação [2].

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

Fig. 2 - Gramática de um programa RS.

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

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

Para representar as possíveis condições de excepção são utilizados os eventos de excepção. Um evento de excepção, 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 excepção associada ao evento. Para activar uma condição de excepção durante uma reacção, 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 excepção *error* e passa os parâmetros *X* e *Y* para o tratamento dessa excepção. A linguagem permite que os eventos de excepção sinalizados em um módulo se propaguem para os outros módulos (todos os mecanismos para esta propagação estão descritos em [12]).

C. 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* [5]. Como o código objecto não é um ficheiro executável, o sistema necessita de um interpretador para a execução do autómato. Além do núcleo reactivo de controle, uma aplicação reactiva requer a implementação de uma interface de 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 [1], Lustre [8] e outras linguagens síncronas, RS não é uma linguagem auto-suficiente; a interface de 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 na execução paralela de todas as regras que possuem a condição de disparo verdadeira. A primeira acção de um passo é uma acção implícita, que desliga todos os sinais contidos nas regras 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 nenhuma regra do programa. Nessa situação, o programa espera até que um novo sinal externo seja fornecido pelo ambiente, o que pode iniciar uma nova reacção (uma sequência de passos) [11].

D. 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.

```
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:=count-1,
                 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 - Autómato gerado para o rato.

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

Fig. 5 - Regras para o autómato da figura 4.

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 autómato RS gerado pelo compilador é apresentado em duas partes: o autómato propriamente dito e as acções a serem executadas por este autómato. Na representação do autómato (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 sequência de acções a executar quando, no estado *n*, ocorrer o sinal *s*. Na lista, as acções são indicadas por números que identificam regras de execução (figura 5) Os asteriscos separam os trechos correspondentes aos passos de execuções referidos anteriormente e toda sequência termina com *go_to(k)*, onde *k* é o próximo estado do autómato.

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

As regras de execução, descritas 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 acções do programa e não são mais necessárias em tempo de execução. Convém observar que, neste exemplo, as regras 1 e 2 não têm acções para executar; elas poderiam ser eliminadas, desde que fossem eliminadas as referências a elas, no autómato.

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 acções. A transição mais complexa do autómato 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 acções da regra 4-1 e continua no estado 2; caso contrário, se a segunda condição da regra 4 é verdadeira, então executa as acções da regra 4-2 e vai para o estado 1.

III. 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 defini-los como sinais internos do programa RS;
- especificar um sinal de entrada especial para dar início a execução do autómato, e consequentemente, do programa a ser executado pelo robô. Este sinal corresponde ao estado *pronto* do robô, estado em que o mesmo se encontra

ligado e com o programa a ser executado em sua memória;

- identificar que operações deverão ser executadas pelo robô para realizar determinada tarefa correctamente e defini-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 de se preocupar com as coordenadas reais dos mesmos, pois assume-se cada um representado por um identificador, podendo supor que em algum outro momento esses identificadores serão mapeados para as respectivas coordenadas. Os pontos podem ser resgistados e mapeados antes ou depois de feito o programa RS, utilizando-se, por exemplo, um *teach box*.

Em relação à eficiência do código gerado, queremos deixar claro que nos vários testes práticos realizados (que incluem os dois exemplos das subsecções seguintes) se verificou que o programa executava sem qualquer degradação de performance relativamente aos correspondentes programas escritos à mão. Embora sendo exemplos de complexidade reduzida, podemos dizer que não tivemos problemas, nem a nível de memória nem de tempo de execução.

A. Exemplo de programa RS para controlar um robô industrial

Um exemplo de programa escrito em RS e utilizado, na prática, em um robô Nachi SC15F pode ser visto na figura 6 [10].

```
rs_prog nr_0001:
[input: [ligado, sensor],
 output: [ir(P), fim, ferramenta(C),
          usar(B), deslocar(D,X,Y,Z)],
 module mov:
 [ input: [ligado, 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)],
  ligado#[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

Uma vez ligado, o sinal externo *ligado* permanece activo até que seja emitido o sinal *fim* para o exterior.

Sejam *P1, P2* e *P3* três pontos de coordenadas conhecidas e fixas, bem como *P4, P5* e *P6*. O robô, quando ligado, passa de uma posição inicial qualquer para o ponto *P1*, onde inicia um ciclo normal: de *P1*, o robô se dirige para o ponto *P2*; de *P2* o mesmo faz um deslocamento, em linha recta, para as coordenadas definidas explicitamente por *X, Y, Z*, mantendo a orientação da ferramenta; deste ponto, o robô se dirige para o ponto *P3*; de *P3* o mesmo faz novamente um deslocamento, em linha recta, para as coordenadas definidas explicitamente por *X′, Y′, Z′*, mantendo a orientação da ferramenta; e deste ponto retorna para o ponto *P1*. Este ciclo se repete por dez vezes.

Quando o sensor *sensor* for accionado, e se o robô se encontrar em *P3*, este assumirá outro comportamento, devendo passar pelos pontos *P4, P5* e *P6*, e assim permanece enquanto esse sinal se mantiver. Desactivado o sensor, após *P6* o robô retorna a *P1*.

Quando compilado, este programa RS irá originar um autómato, cujas características já foram descritas na secção 2.D. Em [10] foi desenvolvido um tradutor capaz de, a partir do autómato 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.

```
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.

Este tradutor tornou possível a utilização da linguagem RS no desenvolvimento de programas para robôs específicos Nachi SC15F, permitindo assim abstrair dos detalhes concretos do equipamento e escrever mais rápido e facilmente um programa de alto nível.

B. Exemplo de programa RS para controlar um robô móvel

Outro exemplo é o programa construído em RS para simular o controle de um veículo móvel, chamado Tó [6], o qual deve vaguear evitando obstáculos. Caso ocorra o aparecimento de algum objecto móvel de interesse, o Tó deve seguí-lo. Quando o objecto 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 recto, para o motor de direcção, 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 modelação baseada em comportamentos, no caso a Arquitectura de Subsunção [9]. Foram assim adicionados na linguagem RS os mecanismos de controle: inibidores e supressores [4].

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 activo. Desta forma, enquanto o sinal dominante estiver presente, o módulo em questão não emitirá o sinal inibido.

Os supressores possuem a sintaxe semelhante à 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 activo. 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.

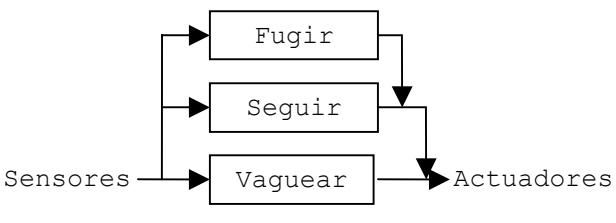

Fig. 8 - Decomposição do Tó baseada na Arquitectura de Subsunção.

Aplicando a modelação baseada na Arquitectura de Subsunção ao desenvolvimento do Tó, o programa é decomposto em módulos conforme mostrado na figura 8. Posteriormente, cada um dos módulos é desenvolvido separadamente; terminados todos os testes individuais, os módulos são unidos utilizando-se mecanismos inibidores e/ou supressores, compondo, desta forma, o sistema final.

Em uma versão anterior, apresentada em [3], os mecanismos de controle eram implementados pelo próprio programador, durante o desenvolvimento do sistema. Com esta 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 dos referidos mecanismos. Isto torna os módulos independentes uns dos outros (caixas pretas), sendo cada um responsável apenas por seu comportamento específico (com respectiva emissão de sinais), o que facilita a construção de sistemas mais complexos.

```
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.

```
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.

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*, para o ambiente externo. Quando o sinal *tick* for recebido, será activado um sinal temporário *next1*, que terá como parâmetro um número inteiro aleatório *N*. Este sinal temporário é responsável por disparar a regra que controla a direcção 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 activar 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 10 é responsável por fazer o Tó perseguir algum objecto de interesse. Este módulo recebe como entrada apenas o sinal externo *sIV*, porém emite dois sinais de saída directamente para a camada de interface com o ambiente externo, *s_andar* e *s_virar*, responsáveis pelo accionamento dos motores de movimento e de direcção, respectivamente. Desta maneira, quando o parâmetro *S*, que indica a intensidade dos sensores dianteiros, valer 1, será disparada a regra que decide a direcção e o sentido a ser tomado pelo Tó para perseguir o objecto. A perseguição é dirigida pelos valores dos parâmetros *D1* e *D2*. Além dos comandos para perseguir o objecto, são emitidos comandos para suprimir os sinais que possam vir a ser emitidos pelo módulo vaguear, por um determinado tempo, no caso 3 unidades de tempo. O ambiente de execução será responsável pelo correcto funcionamento dos mecanismos de controle

No módulo apresentado na figura 11, responsável pelo comportamento de fuga do veículo móvel, faz com que o mesmo evite obstáculos. Este módulo recebe como entrada apenas o sinal externo *sIV*, como na versão anterior. No entanto, emite dois sinais de saída, *f_andar* e *f_virar*. 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.

A figura 12 contém as declarações para o ambiente de execução que 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. Neste exemplo, será usado o tipo criado na extensão, 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 activo. 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 Arquitectura de Subsunção que são a modularidade, simplicidade, independência e robustez, são incorporadas no sistema, uma vez que cada módulo é responsável apenas por um comportamento, o que o faz com eficiência e segurança.

IV. CONCLUSÃO

Este artigo teve como objectivo 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 abstracção para a programação de robôs. Sendo

uma linguagem declarativa e de mais alto nível conceptual do que as tradicionais linguagens imperativas, próximas da linguagem assembly, usadas na programação de robôs, a RS permite realizar essa tarefa em menos tempo e com menor esforço, simplificando ainda a manutenção dos programas. Para isso contribui a capacidade de distribuir tarefas oferecidas pela RS e o seu nível de modularidade.

Com uma linguagem declarativa e modular ganha-se não só em termos de manutenção, mas também, claramente, ao nível da reutilização de programas, o que se traduz em aumento de fiabilidade e decréscimo no tempo de desenvolvimento.

Defendemos, ainda, que o uso de uma linguagem de alto nível como a RS, que pode ser compilada para código de máquina, aumenta a portabilidade dos programas uma vez que a mesma especificação pode ser usada, sem alterações, para gerar código para robôs diferentes. Basta que se desenvolvam tradutores específicos para cada equipamento, tal como já sucede há anos com as linguagens de programação e os diversos processadores.

Um estudo mais aprofundado está a ser realizado para identificar as características requeridas a 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, permitindo descrever, não só o comportamento do robô, como também a sua integração com o resto da célula e as comunicações com o exterior.

Não tendo sido preocupação das experiências realizadas o estudo da eficiência do código gerado, impõe-se agora, como trabalho futuro, o desenvolvimento de uma série de outros testes que permitam concluir da qualidade do código produzido automaticamente a partir do autómato e aferir da necessidade de proceder à sua optimização.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", Science of Computer Programming, 19, 87-152, 1992.

[2] G. R. Librelotto, "Um Compilador para a Linguagem RS Distribuída, Dissertação de Mestrado, PPGC, UFRGS, Porto Alegre, Brasil, 2001.

[3] G. V. Arnold and S. S. Toscani, "Using the RS Language to Control Autonomous Vehicles", Workshop on Intelligent Components for Vehicles, International Federation of Automatic Control, Seville, Spain, 465-470, 1998.

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

[5] J. E. Hopcroft and J. D. Hullman, "Introduction to Automata Theory, Languages and Computation", Addison-Wesley, Massachussetts, USA, 1979.

[6] L. Corrêa, "O Veículo de Três Olhos (Tó para os amigos)", Publicação interna. Lisboa: Departamento de Robótica, Universidade Nova de Lisboa, 1992.

[7] M. P. Groover, "Automation, Production, Systems, and Computer-Integrated Manufacturing", Prentice Hall, Inc., 1987.

[8] N. Halbwachs, "Synchronous Programming of Reactive Systems", Klumer Academic Publisher, Dordrecht, 1993.

[9] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot" , IEEE Journal of robotics and Automation, RA-2, n. 1, 14-23, 1986.

[10] S. J. Piola, "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, 1998.

[11] S. S. Toscani, "Introdução aos Sistemas Reativos", CPGCC, UFRGS, Porto Alegre, Brasil, 1996.

[12] S. S. Toscani, "RS: Uma Linguagem para Programação de Núcleos Reactivos", Tese de doutoramento, Depto de Informática, UNL, Lisboa, Portugal, 1993.