Apresentação da linguagem reativa síncrona RS

S. S. Toscani e L. F. Monteiro

Sumário

O artigo introduz a linguagem reativa síncrona RS, a qual é apropriada para a programação de sistemas que devem responder, de forma eficiente, a sinais provenientes de um ambiente externo. Os programas são formados por regras de reação do tipo *condição* ⇒ *ação* que são postas em execução por sinais vindos do exterior. A linguagem origina programas concorrentes claros e concisos, os quais são compilados para autômatos finitos determinísticos. Os problemas de escalonamento e sincronização são resolvidos em tempo de compilação.

Abstract

This paper is an introduction to the reactive synchronous programming language RS which is suited for applications where the computer has to answer efficiently to externally generated signals. The programs are formed by reaction rules of the type *condition* ⇒ *action* which are triggered by signals arriving from an external environment. The language produces clear and concise concurrent programs which are compiled to deterministic finite automata. The scheduling and synchronization problems are treated at compile time.

Simão Sirineo Toscani
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Caixa Postal 15064
91501-970 Porto Alegre - RS
Brasil
e-mail: simao@inf.ufrgs.br

Luís Fernando Monteiro
Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
Portugal
e-mail: lm@fct.unl.pt

1 Introdução

A expressão "sistema reativo" foi introduzida por D. Harel e A. Pnueli [9] para denominar os sistemas que, continuamente, devem responder a estímulos provenientes de um ambiente externo, numa ordem desconhecida. Harel e Pnueli propuseram uma nova dicotomia para separar sistemas facilmente tratáveis de sistemas problemáticos. Segundo eles, a relação de dicotomias clássicas (sistemas determinísticos × sistemas não determinísticos, sistemas seqüenciais × sistemas concorrentes, etc) poderia ser aumentada com uma outra que melhor separava os sistemas relativamente fáceis de desenvolver dos sistemas difíceis. A nova dicotomia proposta foi: sistemas transformacionais × sistemas reativos.

Um *sistema transformacional* computa resultados a partir de um conjunto de dados. Exemplos típicos são compiladores e programas para solucionar problemas numéricos. Também são considerados transformacionais os sistemas que requisitam entradas adicionais e/ou produzem parte de suas saídas no decorrer das suas execuções.

Os *sistemas reativos*, por outro lado, caracterizam-se por interagir fortemente com um ambiente, mantendo um relacionamento dinâmico com esse ambiente. Exemplos de sistemas que repetitivamente reagem a estímulos provenientes do exterior encontram-se em toda parte: controladores de processos industriais, interfaces de usuário, video-games, máquinas de venda automática, relógios digitais com múltiplas funções, etc.

Gérard Berry [1] prefere excluir da categoria de reativos, os sistemas que interagem à sua própria velocidade[1] com usuários ou outros programas. Estes sistemas são denominados *interativos*. Um sistema "time-sharing", por exemplo, é interativo, pois é ele que determina o ritmo das interações com o ambiente externo. Normalmente, os sistemas interativos consomem grande parte do seu tempo envolvidos em comunicação. Berry reserva a denominação "reativo" para os sistemas que interagem num ritmo que é determinado pelo ambiente externo, e cujo objetivo é garantir tratamento acurado para as interrupções a que são submetidos. Estes sistemas, normalmente, não se envolvem em problemas de comunicação.

A denominação *sistema de tempo real* tem sido usada para os sistemas que controlam processos externos que dependem fundamentalmente dos tempos de resposta do computador. Um sistema de tempo real recebe interrupções do ambiente externo e deve gerar comandos para atuar nesse ambiente, dentro de limites estritos de tempo. Os sistemas de tempo real, em geral, são reativos. O recíproco, contudo, não é verdadeiro. São comuns programas reativos que não são considerados como de tempo real. Exemplos são interfaces de usuários, protocolos de comunicação e "drivers" de sistemas operacionais.

Neste artigo, a denominação "sistema reativo" será usada no sentido mais restrito, conforme caracterizado por G. Berry. Estes sistemas normalmente são organizados de maneira especial e requerem um estilo próprio de programação. A organização natural para um sistema reativo envolve três camadas:

- Uma camada de *interface* com o ambiente, a qual se encarrega da recepção dos estímulos e do encaminhamento das saídas. Esta camada manipula interrupções, lê sensores e dispara atuadores. Ela transforma eventos físicos externos em sinais lógicos internos e vice-versa.
- Um *núcleo reativo*, que contém a *lógica* da execução e constitui a parte central e mais difícil do sistema. Esta camada manipula entradas e saídas lógicas, e realiza as reações: para cada entrada, efetua computações e gera saídas.
- Uma camada de *manipulação de dados*, que executa as computações triviais[2] requeridas pelo núcleo reativo.

Esta organização é natural, visto que, normalmente, numa aplicação reativa, o módulo de controle fica separado do restante do sistema. Por outro lado, esta organização permite que os núcleos reativos sejam estudados, projetados e implementados de forma completamente separada do ambiente externo com o qual interagem. Esta separação do mundo físico torna os núcleos reativos muito atrativos para o desenvolvimento de estudos teóricos.

Dentre as ferramentas que têm sido utilizadas para programar sistemas reativos, duas se destacam: autômatos determinísticos e linguagens concorrentes. Os primeiros têm sido usados para programar núcleos reativos de pequeno porte, tipicamente em protocolos ou

[1] Isto é, sem exigências do ambiente externo, em relação a tempo de resposta.

[2] Computações clássicas, normalmente implementadas por procedimentos externos escritos numa outra linguagem.

controladores. Os autômatos permitem obter excelentes (e mensuráveis) desempenhos em tempo de execução e apresentam a vantagem de serem bem conhecidos matematicamente. Provas de correção não triviais podem ser efetuadas por verificadores automáticos de fórmulas da lógica temporal, tais como EMC [5] e CESAR [13], ou por sistemas de observação de autômatos. como AUTO [15]. Porém, o projeto e manutenção de autômatos é tarefa difícil e muito sujeita a erros. Pequenas mudanças na especificação podem acarretar profundas mudanças nos autômatos. Acima de tudo, os autômatos são puramente seqüenciais e não suportam concorrência: a combinação de autômatos independentes (concorrentes) em um único autômato é tarefa muito complexa.

Linguagens de programação concorrente são ferramentas mais elaboradas. Normalmente, elas permitem o desenvolvimento hierárquico e modular de programas. Os mecanismos para controle de processos e as primitivas de comunicação são definidos ao nível da linguagem e, portanto, são portáteis. Freqüentemente são providos recursos para definição de interfaces e manipulação de dados, o que permite a programação, em uma única linguagem, das três camadas referidas anteriormente. Contudo, todas as linguagens concorrentes clássicas são não determinísticas. A semântica das primitivas para manipulação do tempo é vaga e imprecisa. O "overhead" de execução pode ser importante e os tempos de execução são imprevisíveis.

Seria conveniente poder unir a eficiência de execução dos autômatos com as facilidades de programação das linguagens concorrentes. O caminho para essa união é a adoção da *hipótese do sincronismo*, a qual dá origem aos *sistemas reativos síncronos*. A hipótese se resume em supor que cada reação seja instantânea — e, portanto, atômica. A reação em tempo zero torna os sinais de saída síncronos com os sinais de entrada. Esta hipótese equivale a supor que o processador encarregado de executar a reação não gasta tempo com seqüenciação de instruções, controle e intercomunicação de processos nem manipulação básica de dados (e.g. adições). Por outro lado, equivale a dizer que o ambiente externo permanece inalterado ou "congelado" durante a reação.

As *linguagens reativas síncronas* permitem a manipulação precisa do tempo e, por outro lado, conciliam concorrência e determinismo. A manipulação do tempo através de mecanismos bem definidos semanticamente é consequência da hipótese do sincronismo. Esta hipótese permite considerar a passagem do tempo como um evento sinalizado do exterior[3], o que resulta em simplicidade (e clareza) de semântica para os mecanismos de manipulação do tempo.

Para conciliar concorrência e determinismo as linguagens síncronas adotam leis semânticas que impedem a ocorrência de condições de corrida nos programas da linguagem. A ausência de condições de corrida permite escalonar os processos paralelos de forma arbitrária. Como qualquer seqüenciação dos processos leva ao mesmo resultado, isto reduz enormemente o espaço de estados do programa, viabilizando a pesquisa exaustiva desse espaço e tornando possível a representação de programas síncronos por autômatos finitos.

[3] E, como consequência, permite não considerar a sinalização de outros eventos durante o tratamento de um estímulo.

2 A linguagem RS

A hipótese do sincronismo corresponde a considerar sistemas ideais, que reagem instantaneamente a cada estímulo externo com uma transformação de estado interno e com uma emissão de sinais. A linguagem RS [14] é uma notação adequada para representar este tipo de comportamento, pois um programa é uma especificação quase direta das transformações internas e das emissões de sinais que devem acontecer para cada possível estímulo.

Um programa RS deve ser visto como se as suas ações internas fossem executadas por um processador infinitamente rápido, em um tempo infinitesimal[4]. O tempo só passa para o ambiente externo e a comunicação de que transcorreu algum tempo é efetuada por um sinal de entrada normal (declarado como qualquer outro sinal do programa). Como as operações internas não consomem tempo, os tempos de resposta satisfazem automaticamente as exigências externas e não é necessário se preocupar com a ocorrência de outros sinais durante a execução de uma reação. Isto simplifica a tarefa do programador, evidentemente.

A hipótese do sincronismo é condizente com a suposição de *estímulos simples*, constituídos por um único sinal: como, eletronicamente, os sinais são sempre detectados em sequência, separados por algum intervalo de tempo, e como não há demora na resposta para um sinal, é natural receber e tratar um sinal por vez. A suposição de que um programa seja estimulado por um único sinal de cada vez é a segunda hipótese básica de RS.

Um programa RS trabalha com variáveis clássicas, que são compartilhadas a nível de módulo, e com *sinais* que são utilizados para comunicação com o exterior e para sincronização interna. Os sinais são identificados por nomes, como *s*, *tick*, etc e podem conter valores ou não. Se um sinal não contém valor ele é dito *puro*. A notação $s(V1, V2)$ indica que o sinal s contém os valores $V1$ e $V2$.

Os sinais são particionados em: *sinais de entrada*, *sinais de saída* e *sinais internos*. Os primeiros são os únicos que podem desencadear reações; os segundos são utilizados para comunicar os resultados das reações; os últimos são usados para sincronização/comunicação interna de processos. Além disso, existem sinais de entrada especiais, denominados *sensores*, os quais não possuem poder de interrupção, nem podem ser puros. Seus valores estão sempre acessíveis e podem ser utilizados em qualquer reação[5].

Um controlador de metrô [4] permite ilustrar os tipos de sinais utilizados num programa. O controlador pode receber um sinal a cada milisegundo, um sinal a cada revolução de roda, sinais de trilho conduzindo informações posicionais, e sinais provenientes do console do operador; o controlador pode usar sensores para medir a temperatura externa e pode emitir comandos para motores e freios. Um sub-módulo do controlador pode receber e emitir sinais adicionais gerados por software, para se comunicar e sincronizar com outros sub-módulos.

Um programa RS é formado por um conjunto de módulos, cada módulo é formado por um conjunto de caixas e cada caixa é formada por um conjunto de regras de reação. Os

[4]Tanto faz considerar a linguagem como formada por comandos ideais, que não gastam tempo de execução, como considerar que os comandos sejam reais, porém executados por um processador ideal infinitamente veloz; o que importa é que o ambiente externo permaneça inalterado durante o tempo da reação.

[5]Supõe-se que a interface com o ambiente atualize automaticamente os valores dos sinais que representam sensores, antes de desencadear uma reação.

módulos e as caixas permitem estruturar o programa, mas, a rigor, não são necessários: qualquer programa pode ser especificado por um único conjunto de regras.

A linguagem pode ser usada em quatro níveis ou versões, denominadas RS-0, RS-1, RS-2 e RS plena. Na versão básica RS-0, a linguagem oferece um conjunto mínimo de mecanismos e os programas são formados por um único conjunto de regras de reação. Na versão RS-1, os recursos de programação são aumentados, mas um programa continua a ser formado por um único conjunto de regras. Na versão RS-2, os programas passam a ser melhor estruturados, pois as regras são agrupadas em caixas, as caixas são agrupadas em módulos e os módulos (um ou mais) é que formam os programas. Na versão plena, a linguagem é estendida para implementar o tratamento de eventos de exceção. Este artigo descreve apenas a versão básica RS-0. Maiores detalhes sobre RS, inclusive comparações com outras linguagens, podem ser encontrados em [14].

3 A linguagem RS-0

A linguagem RS-0 pode ser vista como o núcleo de RS. Um programa RS-0 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. 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).

Uma regra de reação tem a forma

$$F \Longrightarrow A,$$

onde F é uma *condição de disparo* e A é uma *ação*.

Uma condição de disparo F é uma lista não vazia de sinais internos e/ou de entrada, com no máximo um sinal de entrada. Se a lista contém um sinal de entrada, a regra é dita *global* ou *externa*, caso contrário é dita *local* ou *interna*.

Um *comando simples* é uma sequência

$$[c_1, c_2, \ldots, c_n] \qquad (n \geq 0)$$

cujos elementos são comandos de atribuição ou emissão de sinais.

Uma ação A pode ser um comando simples ou ter a forma:

$$\begin{array}{l} \mathit{case} \\ [\, b_1 \longrightarrow C_1, \\ \quad \ldots, \\ \quad b_n \longrightarrow C_n \,] \end{array} \qquad (n \geq 2)$$

onde $b_1, \ldots, b_n$ são condições booleanas e $C_1, \ldots, C_n$ são comandos simples.

As ações (e regras) do tipo *case* são ditas *condicionais* e as demais, *incondicionais*. Nas regras condicionais, cada $b_i \longrightarrow C_i$ $(i = 1, \ldots, n)$ representa uma opção de escolha para tempo de execução.

A ação de uma regra pode ser vista como um processo seqüencial adormecido à espera do disparo da regra. Se a regra é condicional, o processo tem vários corpos (sequências de instruções) alternativos.

A semântica lembra muito as redes de Petri, pois a execução se desenvolve em passos seqüenciais nos quais as regras com condição de disparo verdadeira são executadas (disparadas) em paralelo.

A condição de disparo de uma regra é verdadeira quando todos os seus sinais (que podem ser vistos como semáforos) estão ligados ou abertos. Quando a condição é verdadeira, os sinais referidos são automaticamente fechados e a ação da regra é executada. Se a regra é condicional, é escolhida a única opção[6] com condição booleana verdadeira.

Um determinado conjunto de sinais abertos faz disparar simultaneamente todas as regras cujas condições de disparo estejam contidas nesse conjunto, e o disparo faz fechar o subconjunto dos sinais referidos nessas condições de disparo. Note-se que um mesmo sinal pode desencadear o disparo de mais de uma regra.

Não há indeterminismo na execução paralela de regras porque os programas RS são obrigados a compartilhar variáveis e sinais de forma disciplinada (acessos conflitantes a variáveis ou sinais são detectados em tempo de compilação e indicados como erros de semântica).

A execução de um programa se processa, então, numa série de passos, nos quais as regras com sinais abertos são executadas simultaneamente. A execução pára quando os sinais abertos não são suficientes para disparar qualquer regra. Neste caso, o programa fica à espera de algum sinal do exterior que venha desencadear uma nova reação.

Pode-se ver um programa RS como um sistema que, ao ser estimulado por algum sinal, se transforma internamente (muda de estado), emite sinais de saída e fica à espera de um novo sinal, tudo instantaneamente e de modo determinístico. Os sinais emitidos e o novo estado interno são dependentes das regras que disparam na reação. Como as reações não consomem tempo, o programa está permanentemente à espera de algum estímulo externo. Em cada estado de espera, diferentes regras ficam habilitadas para disparar, em função do sinal externo que ocorrer. Esta representação de programas é conveniente, pois parece haver concordância geral de que estados e eventos são um meio natural para descrever o comportamento dinâmico de sistemas complexos [8].

Um exemplo simples é mostrado a seguir. O programa usa dois sinais de entrada, $name(X)$ e $password(Y)$, dois sinais internos $gotName(X)$ e $gotPassword(Y)$, e um sinal de saída $checkLogin(X,Y)$. Não são utilizadas variáveis compartilhadas tampouco ação de inicialização. Nas declarações, os campos de valor de um sinal são identificados por nomes que iniciam com letra maiúscula, e mesmos nomes de campos podem ser usados para diferentes sinais.

O programa emite para o exterior o sinal *checkLogin*, sempre que os elementos de um par *name, password* são especificados, em qualquer ordem.

input : $name(X), password(Y)$
output : $checkLogin(X,Y)$
signal : $gotName(X), gotPassword(Y)$
$name(X) \Longrightarrow gotName(X)$
$password(Y) \Longrightarrow gotPassword(Y)$
$gotName(A), gotPassword(B) \Longrightarrow checkLogin(A,B)$

Os sinais declarados como input só podem ser ligados pelo ambiente externo (um sinal

[6]Garantidamente, existe uma e só uma condição que é verdadeira, qualquer que seja o estado de execução do programa.

por vez). Os sinais declarados como signal ou output são ligados pelas ações das regras. Quando um sinal é ligado, as suas variáveis são instanciadas.

A primeira regra de reação diz que o estímulo $name(X)$ deve ligar o sinal interno $gotName$, instanciando o único campo de valor desse sinal com o valor X recebido.

A segunda regra diz que o estímulo $password(Y)$ deve ligar o sinal interno $gotPassword$, instanciando o seu campo de valor com Y.

A terceira regra diz que, quando forem ligados os sinais $gotName$ e $gotPassword$, então, chamando o valor associado ao primeiro de A e o valor associado ao segundo de B, deverá ser emitido o sinal $checkLogin(A, B)$.

Numa condição de disparo, as variáveis que representam os valores dos sinais podem ser vistas como *argumentos de entrada* para o processo correspondente à ação da regra. Cada regra (processo) atua sobre um conjunto privativo de variáveis, as quais são instanciadas através da condição de disparo da regra.

4 A linguagem RS-A

A semântica de RS-0 é formalizada através de uma linguagem abstrata denominada RS-A. Aqui é apresentada a versão da linguagem abstrata que se preocupa apenas com os aspectos de controle determinados pela recepção e envio de sinais, ignorando a manipulação de valores em sinais ou em variáveis internas. Tal linguagem é denominada linguagem RS-A de controle ou simplesmente RS-C.

4.1 Sintaxe da linguagem RS-C

Supõem-se dados três conjuntos não vazios de "sinais", disjuntos dois a dois:

- In, de *sinais de entrada*;
- Out, de *sinais de saída*; e
- Sig, de *sinais internos*.

Um *programa* em RS-C é um conjunto de *regras de reação* da forma

$$F \Rightarrow A,$$

em que F é uma *condição de disparo* e A uma *ação*.

Uma condição de disparo F é um subconjunto finito não vazio de $\mathsf{In} \cup \mathsf{Sig}$, contendo no máximo um elemento de In. Se $F \cap \mathsf{In} \neq \emptyset$, a regra de reação diz-se *global* ou *externa*, de contrário *local* ou *interna*.

Uma ação A é um subconjunto finito—possivelmente vazio—de $\mathsf{Sig} \cup \mathsf{Out}$.

4.2 Semântica operacional da linguagem RS-C

A caracterização da semântica operacional da linguagem RS-C segue o modelo proposto por Plotkin [12] e Hennessy [10] conhecido pela designação de semântica operacional estrutural.

A semântica operacional de um programa Π consiste em definir dois sistemas de transição para descrever o comportamento de Π. O sistema de transição $\mathcal{R}_\Pi$ descreve o comportamento "interno" de Π durante uma reação, o qual é "invisível" para o "exterior". O

sistema de transição $\mathcal{T}_\Pi$ descreve o comportamento "global" de Π, que é "observável" do "exterior".

4.2.1 Semântica de uma reação

As configurações do sistema $\mathcal{R}_\Pi$ são os subconjuntos de $\mathsf{In} \cup \mathsf{Sig}$ que têm no máximo um elemento de In. Seja I_Π o conjunto das configurações de $\mathcal{R}_\Pi$. Dada uma tal configuração $S \in I_\Pi$, seja

$$\Pi|S = \{F \Rightarrow A \in \Pi \mid F \subseteq S\}$$

o conjunto das regras de Π cujas condições de disparo estão contidas em S. Uma regra em $\Pi|S$ diz-se que está *habilitada* em S.

As transições deste sistema são etiquetadas por conjuntos de sinais de saída, e a relação de transição $\rightsquigarrow \subseteq I_\Pi \times \wp(\mathsf{Out}) \times I_\Pi$ define-se do seguinte modo:

$$S \overset{Y}{\rightsquigarrow} S'$$

se e só se $\Pi|S \neq \emptyset$,

$$Y = \bigcup_{F \Rightarrow A \in \Pi|S} A \cap \mathsf{Out}$$

e

$$S' = (S - \bigcup_{F \Rightarrow A \in \Pi|S} F) \cup \bigcup_{F \Rightarrow A \in \Pi|S} A \cap \mathsf{Sig}.$$

Assim, para que exista uma transição a partir de uma certa configuração S, é necessário que alguma regra de reação esteja habilitada em S. Se for esse o caso, as regras habilitadas "disparam" em simultâneo, produzindo os sinais de saída em Y. A configuração seguinte obtem-se "desligando" (removendo de S) os sinais que ocasionaram o disparo de regras, e "ligando" (acrescentando a S) os sinais produzidos pelas regras habilitadas. Uma "reação" é uma sequência de transições de $\mathcal{R}_\Pi$ iniciada numa configuração com um sinal de entrada (e, possivelmente, sinais internos) e terminando numa configuração sem nenhuma regra habilitada. Uma definição precisa será dada mais abaixo, visto que uma reação é essencialmente uma transição do sistema $\mathcal{T}_\Pi$.

4.2.2 Semântica do comportamento observável

O sistema de transição $\mathcal{T}_\Pi$ que descreve a reação conforme ela é "vista do exterior" define-se do seguinte modo:

- As configurações de $\mathcal{T}_\Pi$ são os conjuntos $S \subseteq \mathsf{Sig}$ de sinais internos que não habilitam nenhuma regra de reação de Π, isto é, $\Pi|S = \emptyset$. Seja C_Π o conjunto das configurações de $\mathcal{T}_\Pi$.
- A relação de transição é $\longmapsto \subseteq C_\Pi \times \mathsf{In} \times \wp(\mathsf{Out}) \times C_\Pi$. Escreve-se

  $$S \overset{x,Y}{\longmapsto} S'$$

  para $S, S' \in C_\Pi$, $x \in \mathsf{In}$ e $Y \subseteq \mathsf{Out}$, e está definida a seguir.

Uma transição de $\mathcal{T}_\Pi$ representa uma *reação* à chegada de um sinal de entrada x com produção dos sinais de saída em Y. A sua definição é assim feita em termos do sistema de transição $\mathcal{R}_\Pi$. Tem-se $S \xrightarrow{x,Y} S'$ se e só se existirem $S_0, S_1, \ldots, S_k \in I_\Pi$ e $Y_1, \ldots, Y_k \subseteq \mathrm{Out}$ tal que:

- $S_0 = S \cup \{x\}$ e $S_k = S'$.
- $S_i \overset{Y_{i+1}}{\rightsquigarrow} S_{i+1}$ para $0 \leq i < k$.
- $Y = Y_1 \cup \cdots \cup Y_k$.
- Se, para alguns i, j tais que $0 \leq i, j < k$, existem $F \Rightarrow A$ em $\Pi|S_i$ e $F' \Rightarrow A'$ em $\Pi|S_j$ com $A \cap A' \neq \emptyset$, então $F = F'$, $A = A'$ e $i = j$.

As primeiras três condições são evidentes. Note-se que $\Pi|S_k = \emptyset$, visto que $S_k = S'$ e $S' \in C_\Pi$. A última condição significa que um mesmo sinal, interno ou de saída, não pode ser emitido mais de uma vez durante uma reação. Note-se, em particular, que fica proibida a repetição de uma regra com ação não vazia durante uma reação. mas não se impõe semelhante restrição a regras com ação vazia.

5 As redes reativas síncronas

Conforme já referido, pode-se ver um programa RS como uma espécie de rede de Petri. Esta analogia com as redes de Petri é conveniente, pois isto facilita o entendimento do comportamento dinâmico dos programas. As redes correspondentes aos programas RS são chamadas *redes reativas síncronas* ou, simplesmente, *redes* RS.

Como nas redes de Petri [11], os nós que formam as redes RS são de dois tipos, *lugares* e *transições*, e os arcos só podem conectar nós de tipos diferentes. Os lugares podem receber *marcas*, as quais são representadas graficamente por pontos pretos, e uma dada configuração de marcas dentro de lugares, define uma *marcação* para a rede.

Uma rede marcada é executada fazendo disparar as transições que tenham marcas em todos os seus lugares de entrada. O disparo de uma transição remove uma marca de cada lugar de entrada e coloca uma marca em cada lugar de saída, o que resulta numa nova marcação para a rede.

Uma rede RS difere de uma rede de Petri convencional por apresentar as seguintes características:

- Um lugar pode conter no máximo uma marca.
- Uma marca pode participar no disparo simultâneo de duas ou mais transições.
- As marcas podem armazenar valores arbitrários.
- O disparo de uma transição é acompanhado de uma ação que atua sobre uma memória associada à rede.
- Existem transições condicionais, as quais podem disparar de várias maneiras alternativas (cada opção de disparo está associada a uma condição booleana).
- As execuções são determinísticas.

As duas primeiras características têm um carater meramente de controle. (A segunda, por exemplo, mostra que nas redes RS não existem situações de conflito ou indeterminismo no disparo de transições.) As duas características seguintes mostram que uma rede RS pode manipular valores nas marcas e numa memória associada à rede. Conseqüentemente, podem usar-se transições condicionais, em que a opção escolhida é determinada pelos valores nos lugares de entrada da transição e pelo estado da memória. A última característica é fundamental para o estudo dos sistemas reativos síncronos e permite representar uma rede por um autômato finito.

6 O processo de geração dos autômatos

O autômato correspondente a um programa pode ser obtido através da análise da rede RS que lhe corresponde. O processo de geração é esboçado a seguir.

Chamam-se *estados de espera* os estados da rede no final de cada reação (estados nos quais a rede fica à espera de estímulos do exterior). Nesses estados, nenhuma transição tem os seus lugares de entrada completamente marcados.

Chamam-se *sinais significativos* para um estado de espera os sinais do exterior que são aguardados nesse estado (sinais que podem desencadear uma reação nesse estado).

O algoritmo gerador do autômato percorre e numera os distintos estados de espera da rede, associando cada um deles a um estado do autômato. (Como os lugares são finitos e podem ter no máximo uma marca, o número de estados que uma rede pode assumir é finito).

Supõe-se que a ação de inicialização do programa realize a marcação inicial da rede, originando o *estado de espera inicial* para a mesma.

A geração do autômato consiste, então, em analisar (e memorizar) os comportamentos da rede para todos os pares <*estado de espera*, *sinal significativo*> possíveis, começando pelo estado de espera inicial.

Para cada estado de espera, é verificado como a rede reage para cada um dos sinais externos significativos. A sequência de transições que disparam quando a rede, no estado de espera n, é estimulada por um sinal s, determina a sequência de ações que o autômato deve executar quando, no estado n, é estimulado por s. Esta sequência é formada por subsequências ou trechos que correspondem a passos de execução (disparo paralelo de transições). Cada trecho é uma seqüenciação das ações paralelas de um passo (todas as seqüenciações possíveis são equivalentes, como se sabe).

Cada sequência de execução resulta numa nova marcação para a rede e num novo estado de espera a ser examinado. Na realidade, as transições condicionais originam bifurcações nos caminhos de execução e fazem com que, ao invés de sequências, se tenha árvores de execução.

No final, o autômato estará representado por triplos $<n, s, a>$ que indicam, para cada estado n e sinal significativo s, a árvore de execução a que lhe corresponde. A árvore conterá todos os caminhos alternativos possíveis, assim como as condições a serem avaliadas em tempo de execução para decidir pelo caminho correto. A cada caminho corresponderá um próximo estado para o autômato e esse novo estado estará indicado no final do caminho, na folha da árvore.

Em outras palavras, todas as combinações de condições estarão previstas (representadas

na árvore) e cada caminho completo (da raiz à uma folha) representará uma possível reação[7] para o autômato. Obviamente, além das condições booleanas para escolher o caminho correto, deverão estar acessíveis, em tempo de execução, as representações das variáveis e sinais cujos valores podem ser alterados durante as reações.

Embora os valores das variáveis e sinais utilizados em um caminho possam variar durante a reação, é essencial que os trechos correspondentes à seqüenciação das ações paralelas de um passo trabalhem com valores fixos para esses elementos (a semântica exige isto para garantir determinismo). Isto obriga que, na representação do autômato, os caminhos de execução sejam particionados explicitamente em trechos. Em tempo de execução, as regras de cada trecho utilizarão valores únicos de variáveis e sinais, como deve ser.

7 Um exemplo

Protótipos da linguagem RS, nas suas várias versões, foram implementados em "standard" Prolog[8]. Cada protótipo inclui um compilador e um simulador de execução com facilidades de depuração. O programa a seguir é mostrado no formato em que ele é submetido ao compilador da versão RS-2. O autômato também é mostrado conforme ele é listado pelo compilador.

A condição de disparo de uma regra tem a seguinte forma geral:

$$s_e \# [s_1, s_2, \ldots, s_n]$$

onde s_e é um sinal de entrada (opcional) e s_i $(1 \leq i \leq n)$ é um sinal interno. Se a regra é *local* ou *interna* ela tem condição de disparo com a forma: $\#[s_1, s_2, \ldots, s_n]$.

Nas ações das regras pode-se ter chamadas de procedimentos externos e os seguintes comandos:

v := *expression*
up(*internal_signal*)
emit(*output_signal*)

O primeiro é utilizado para atribuir o valor de uma expressão a uma variável, o segundo para ligar um sinal interno, e o terceiro para emitir um sinal externo.

O programa exemplo implementa um diálogo "login" no qual o usuário tem 3 chances para especificar corretamente seu nome e senha, usando os sinais *name*(X) e *password*(Y), em qualquer ordem. O programa emite os sinais *loginSuccessful* ou *loginUnsuccessful* consoante o usuário tenha sucesso ou não. A relação de usuários válidos é representada por fatos Prolog colocados na camada de manipulação de dados do programa (logo após o ponto que identifica o fim do programa RS).

```
rs_prog login:
[ input : [name(X), password(Y)],
  output: [loginSuccessful, loginUnsuccessful],

  module get_login:
```

[7] Cada reação é uma sequência de ações acompanhada de uma mudança de estado.

[8] Como não são utilizados recursos especiais, em princípio qualquer versão do Prolog pode ser utilizada para executar o protótipo. Porém, programas RS muito extensos podem estourar a capacidade de representação de termos de certas implementações, já que um programa é representado internamente como um termo Prolog.

```
 [ input  : [name(X), password(Y), goodLogin, badLogin, tooManyAttempts],
   output : [loginSuccessful, loginUnsuccessful, checkLogin(X,Y)],
   signal : [gotName(X), gotPassword(Y)],
   initially: [nl, write('Enter name and password')],

   name(X) ==> [up(gotName(X))],
   password(X) ==> [up(gotPassword(X))],
   #[gotName(X), gotPassword(Y)] ==> [emit(checkLogin(X,Y))],
   goodLogin ==> [emit(loginSuccessful)],
   badLogin  ==> [nl, write('Please, repeat name and password')],
   tooManyAttempts ==> [emit(loginUnsuccessful)]
 ],

 module check_login:
 [ input   : [checkLogin(X,Y)],
   output  : [goodLogin, badLogin, tooManyAttempts],
   var     : [count],
   initially: [count := 0],

   checkLogin(X,Y) ==> case
                       [ user(X,Y) --> [count:=0, emit(goodLogin)],
                         count>=2  --> [emit(tooManyAttempts), count:=0],
                         else      --> [count:=count+1, emit(badLogin)]  ]
 ]
].

user(sst, aaa).
user(lvt, bbb).

AUTOMATON:
==========
init  -        [1,8,*,go_to(1)]
  1   name     [2,*,go_to(2)]
  1   password [3,*,go_to(3)]
  2   name     [2,*,go_to(2)]
  2   password [3,*,4,*,[9-1,*,5,*,go_to(1)],
                        [9-2,*,7,*,go_to(1)],
                        [9-3,*,6,*,go_to(1)] ]
  3   name     [2,*,4,*,[9-1,*,5,*,go_to(1)],
                        [9-2,*,7,*,go_to(1)],
                        [9-3,*,6,*,go_to(1)] ]
  3   password [3,*,go_to(3)]

RULES:
======
Module get_login:
  1.   [] ==> [nl,write(Enter name and password)]
  2.   [name(A)] ==> [up(gotName(A))]
  3.   [password(A)] ==> [up(gotPassword(A))]
  4.   [gotName(A),gotPassword(B)] ==> [emit(checkLogin(A,B))]
  5.   [] ==> [emit(loginSuccessful)]
  6.   [] ==> [nl,write(Please, repeat name and password)]
```

```
 7.   [] ==> [emit(loginUnsuccessful)]
Module check_login:
 8.   [] ==> [count:=0]
 9.   Case:
9-1. [checkLogin(A,B)] {user(A,B)} --> [count:=0]
9-2. [checkLogin(A,B)] {count>=2} --> [count:=0]
9-3. [checkLogin(A,B)] {else} --> [count:=count+1]
```

O programa utiliza três procedimentos externos (goals Prolog): *nl* e *write*(X), no módulo *get_login*, e *user*(X, Y), na regra condicional do módulo *check_login*. O procedimento *user*(X, Y) retorna *true* se X e Y especificam o nome e a senha de um usuário válido.

O autômato é apresentado em duas partes: *relação das transições* e *relação das ações*. Na representação das transições, 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 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 são os lados direitos dessas regras). Os asteriscos separam os trechos correspondentes aos "passos de execução" referidos anteriormente e toda sequência 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 para cada módulo). Estas ações são executadas automaticamente antes do autômato ser colocado em funcionamento (portanto, o estado inicial real do autômato é o estado 1).

As regras de execução 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 seqüenciar as ações do programa, e não são mais necessárias em tempo de execução.

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.

8 Conclusão

RS é uma linguagem experimental, com sintaxe simples facilmente reconhecível. Ela destina-se à programação de núcleos reativos, que constituem a parte central e mais difícil de um sistema reativo. Tal como outras linguagens síncronas [3, 6, 7], RS não é uma linguagem de propósitos gerais nem tampouco auto-suficiente. As camadas de interface e de manipulação de dados devem ser especificadas em alguma linguagem hospedeira.

Nas experiências efetuadas, a linguagem demonstrou possuir boas qualidades, quais sejam: programas fontes claros e concisos, e programas objetos (autômatos) bastante eficientes. Entre os sistemas programados está um relógio digital de múltiplas funções, exemplo típico de sistema reativo. Relógios digitais são interessantes por possuirem modularidade não trivial e relativa complexidade [2, 8]. Um grande número de comandos são sinalizados por poucas teclas, usando diferentes modos de funcionamento, e várias informações são apresentadas em mostradores, usando diferentes modos de "display".

Embora exista uma relação grande com a linguagem hospedeira Prolog, após o autômato ter sido obtido, ele pode ser facilmente implementado em qualquer outra linguagem. Todas

as informações necessárias para usar o sistema reativo numa aplicação real estão contidas no autômato e nas regras de execução associadas a ele.

Em relação à hipótese do sincronismo, na realidade, o que se assume é apenas que o tempo de reação seja suficientemente curto para distinguir e tratar de forma precisa os eventos de chegada. Na prática, a validade da suposição pode ser avaliada pela medida do tempo máximo de resposta do programa, a qual pode ser facilmente obtida a partir do autômato que o representa. Como o código correspondente a cada transição é linear (sem "loops"), o tempo máximo de execução pode ser determinado com precisão, para uma dada máquina.

A obtenção do autômato de um programa através da exploração exaustiva do espaço de estados é um processo de compilação aceitável na prática[9]. A par de permitir a verificação de propriedades comportamentais nessa varredura completa dos possíveis estados, a qualidade do código objeto resultante é excelente. Muitas otimizações são realizadas durante o processo ção; a intercomunicação de processos, em particular, é toda realizada nesse tempo, praticamente sem geração de código, e a intercomunicação via diálogos instantâneos através de sinais puros não produz qualquer código. Após a geração do código objeto, mais testes podem ser realizados sobre o autômato resultante, utilizando sistemas de verificação como os citados na seção de introdução.

Em suma, mesmo que não seja completamente verdadeiro no mundo real, o sincronismo é um ótimo paradigma: permite melhor programação, melhor geração de código e melhor verificação de programas.

Embora no seu estado atual a linguagem RS possa ser considerada como tendo boas qualidades para os fins a que se destina, é natural que hajam aperfeiçoamentos para serem introduzidos. No futuro próximo alguns melhoramentos já definidos deverão ser introduzidos. Também pretende-se escrever um novo compilador em C, assim como criar um ambiente de execução mais apropriado para a simulação e teste de programas e, ainda, utilizar a linguagem no desenvolvimento de aplicações de maior porte. As aplicações práticas poderão indicar novos aperfeiçoamentos, os quais serão introduzidos em versões subseqüentes. As experiências realizadas até aqui permitem esperar que RS venha a se constituir, de fato, numa ferramenta útil para o projeto e a construção de sistemas reativos.

Referências

[1] G. Berry. "Real Time Programming: Special Purpose or General Purpose Languages". *Research Report 1065*, INRIA, 1989.

[2] G. Berry. "Programming a Digital Watch in Esterel V3". *Research Report 1032*, INRIA, 1989.

[3] G. Berry and G. Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". *Science of Computer Programming*, 19(2):87-152, 1992.

[4] G. Berry, P. Couronné and G. Gonthier. "Synchronous Programming of Reactive Systems: an Introduction to Esterel". *Research Report 647*, INRIA, 1987.

[9] Para uma linguagem assíncrona, o indeterminismo relacionado ao escalonamento das ações acarreta uma explosão no número de estados, o que inviabiliza o uso desta técnica para tais linguagens.

PDF compression, OCR

[5] E. M. Clarke, E. A. Emerson and A. P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". Carnegie-Mellon University, 1983.

[6] P. Le Guernic, M. Le Borgne, T. Gauthier and C. Le Maire. "Programming real time applications with SIGNAL". *Research Report* 1446, INRIA, 1991.

[7] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. "The Synchronous dataflow programming Language LUSTRE. In *Proceedings IEEE*, volume 79, pages 1305-1320, September, 1991.

[8] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, Vol 8, 1987, 231-274.

[9] D. Harel and A. Pnueli, "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol F13, Edited by K. R. Apt, Springer-Verlag Berlin Heidelberg, 1985.

[10] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley Press, 1990.

[11] J. Peterson. "Petri Nets". *Computing Surveys*, Vol. 9, No. 3, September 1977, pages 223-252.

[12] G. D. Plotkin. "A Structural Approach to Operational Semantics". *Technical Report* DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

[13] J-P. Queile and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In *Proceedings International Symposium on Programming*, Lecture Notes in Computer Systems 137, Springer-Verlag, New York, 1982.

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

[15] D. Vergamini. "Verification by Means of Observational Equivalence on Automata". *Research Report* 501, INRIA, 1986.