UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Um Compilador para a Linguagem RS Distribuída

por

GIOVANI RUBERT LIBRELOTTO

Dissertação submetida à avaliação, como requisito parcial
para a obtenção do grau de Mestre em
Ciência da Computação

Prof. Dr. Simão Sirineo Toscani
Orientador

Porto Alegre, abril de 2001.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Librelotto, Giovani Rubert

Um Compilador para a Linguagem RS Distribuída / por Giovani Rubert Librelotto. - Porto Alegre: PPGC da UFRGS, 2001.
89 f.:il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 2001. Orientador: Toscani, Simão Sirineo.

1. Sistemas Reativos. 2. Linguagem RS. 3. Distribuição. 4. MDX. 5. Núcleos Reativos. I. Toscani, Simão Sirineo. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação : Prof. Philippe Olivier Alexandre Navaux
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Agradecimentos

A todos os funcionários que, prestando seus serviços, contribuíram para que todo o Instituto de Informática funcionasse corretamente.

À Universidade Federal do Rio Grande do Sul e ao CAPES, que me possibilitaram, além da gratuidade no estudo, um auxílio indispensável por meio da bolsa de mestrado das qual dependo há dois anos.

Aos professores, pela forma como buscam novos conhecimentos e pela incansável dedicação.

À aqueles colegas com quem partilhei os principais momentos de minha vida acadêmica: Marco Antônio de Castro Barbosa, Audrei Rossato e Éder Contri. Vocês estiveram sempre comigo desde o trote na faculdade, lá em março 1995, até agora, e tenho certeza que cada um sabe da amizade e gratidão que sinto por vocês. Tenho o maior orgulho em dizer que me formei com vocês e que se não fosse por vocês, galera, não sei se chegaria neste momento que me encontro agora.

Aos amigos semânticos: Ronnie, Simone, Isabel (Bel), Luciana (Lu), Márcia, Marco e Silvana, pelas reuniões, pelas festas e pelo carinhoso companheirismo. Pena não termos aproveitado ainda mais esse tempo de convivência, mas vocês podem ter certeza que ficarão para sempre marcados na minha memória.

À família Ortiz, com a qual convivi por ¾ do tempo total desse mestrado e onde aprendi muito, recebi amor e afeto, e eles sabem que me considerei como um filho deles. Grande Athos Ortiz, você é o amigo que todos gostariam de ter.

Não poderia deixar de falar dos meus amigos de fé, aqueles com quem convivo desde a minha infância, com os quais sei que poderei contar em todas as horas. Guigo, Jairo, Tiago, Oberdan, Zando, Fernando, meu mano Ogi e os que aqui não citei, mas sabem da minha amizade por vocês.

Aos meus amigos e amigas virtuais, do ICQ e do mIRC, com quem passei horas e mais horas teclando nas noites deste ano que passou. Obrigado por me ajudarem, quando eu não conseguia mais ver este trabalho na minha frente, e me darem novo ânimo para continuar essa empreitada.

Ao Prof. Dr. Simão Sirineo Toscani, não tenho palavras. O senhor já estava se aposentando da UFRGS e, mesmo assim, decidiu me orientar. Serei eternamente grato ao senhor e tentarei nunca decepcioná-lo. Tive uma sorte incrível em poder trabalhar consigo, e espero ter satisfeito as suas expectativas.

À minha família, pela educação, pelo apoio e pela compreensão. Cada um, a seu modo, foi essencial não só durante esse período, mas durante toda a minha existência. Sei que estive muito ausente nestes anos, passando, às vezes, longos períodos sem os visitar, mas podem ter certeza de que amo muito vocês e sei que estão tão felizes, nesta hora, quanto eu.

Sumário

Lista de Figuras 7

Lista de Tabelas 8

Lista de Abreviaturas...9

Resumo...10

Abstract 11

1 Introdução 12

2 Os Sistemas Reativos e a Linguagem RS 15

2.1 Sistemas reativos..15
2.2 A linguagem RS 17
2.2.1 Hipótese do sincronismo 18
2.2.2 Sinais 18
2.2.3 Variáveis..19
2.2.4 Caixas de regras...19
2.2.5 Exceções...20
2.3 Sintaxe 20
2.4 Algoritmo de execução 21
2.5 Semântica 22
2.6 Autômato RS gerado 23
2.7 Considerações sobre a utilização de autômatos..25
2.8 Conclusões sobre a Linguagem RS 26

3 A Linguagem RS 5.0...27

3.1 Mudanças na sintaxe de RS 5.0...27
3.1.1 Tornando a linguagem RS mais linear 27
3.1.2 O uso de mais de um sinal disparador do ambiente externo 28
3.1.3 Os sinais inibidores 29
3.1.4 O novo formato das condições de disparo das regras de execução......................29
3.1.5 As declarações de Exclusão Mútua e Concomitância 30
3.1.6 A simplificação da especificação das regras de exceção.....................................31
3.2 Mudanças na Semântica de RS 5.0 32
3.2.1 O disparo de mais de um sinal externo no gatilho 32
3.2.2 O uso de sinais inibidores...32
3.2.3 Problemas encontrados com o uso de vários sinais externos no gatilho 33

3.3 A Distribuição da Linguagem RS 34
3.3.1 Modificações sintáticas 35
3.3.2 O comando *machine* 35
3.3.3 O pré-processador RS 38
3.3.4 Saída do RSD 39
3.3.5 Distribuição com o uso de nomes reais de máquinas 40
3.3.6 O ambiente de execução do RSD 41
3.3.7 Comunicação entre autômatos 42
3.3.8 Transmissão de sinais para outros autômatos 43
3.3.9 O protocolo RS 43

4 O Compilador RS 46

4.1 A Construção do Compilador RS 46
4.2 Gerador de Código 47
4.3 O processo de geração dos autômatos 47
4.4 O Código C para a simulação de Autômatos RS Distribuídos 48
4.4.1 Declaração e tipificação de variáveis e sinais 48
4.4.2 Tratamento de asteriscos 49
4.4.3 Representação do autômato em C 49
4.4.4 Tratamento de exceções internas 50
4.4.5 A interface com o usuário 50
4.4.6 Execução do sistema 51
4.4.7 Implementação do protocolo RS 51
4.5 O Formato OC 52
4.5.1 As tabelas 52
4.5.2 O autômato 55
4.8.3 Exemplo de Código OC 57

5 O Ambiente de Execução RS Distribuído 59

5.1 O Ambiente de Programação Paralela MDX 59
5.1.1 Modelo de programação 59
5.1.2 Implementação 60
5.1.3 Núcleo de comunicação 60
5.1.4 Servidores 62
5.2 MDX-RS – Um Novo Núcleo de Comunicação MDX 62
5.2.1 Necessidades do RS Distribuído 62
5.2.2 Funcionamento de um autômato 63
5.2.3 Estrutura do núcleo de comunicação MDX-RS 64
5.2.4 Comunicação no mouse distribuído 68
5.3 A escolha da linguagem C e do ambiente MDX 69

6 Exemplos de aplicações da Linguagem RS distribuída 71

6.1 O Três Olhos 71
6.1.1 Arquitetura do TÓ 71
6.1.2 Descrição dos Módulos 72
6.2.3 Implementação em RS 74
6.2 Auto-estrada do Futuro 76
6.2.1 Descrição 77

6.2.2 Descrição dos Módulos 78
6.2.3 Implementação em RS.. 79

7 Conclusão 83

7.1 Trabalhos Futuros.. 85

Bibliografia.. 86

Lista de Figuras

FIGURA 2.1 – Linha de um autômato gerado por RS 25
FIGURA 2.2 – Máquina de estados correspondente ao programa RS 25
FIGURA 3.1 – Código da lâmpada em RS 4.0 28
FIGURA 3.2 – Código da lâmpada em RS 5.0 28
FIGURA 3.3 – Código de uma roleta de cinema em RS 5.0...30
FIGURA 3.4 – Trecho de código em RS 4.0 31
FIGURA 3.5 – Trecho de código em RS 5.0 31
FIGURA 3.6 – Regras a serem disparadas em um mesmo estado 33
FIGURA 3.7 – Regras a serem disparadas em um mesmo estado 34
FIGURA 3.8 – Cabeçalho de um programa RS distribuído...36
FIGURA 3.9 – Modelo de declaração de uma *machine* em um código RS 36
FIGURA 3.10 – Formato do arquivo de informação distribuída. 39
FIGURA 3.11 – Estrutura do ambiente RSD rodando *mouse* distribuído 42
FIGURA 3.12 – Relevância de sinais para cada módulo 42
FIGURA 5.1 – Visão do Sistema MDX..60
FIGURA 5.2 – Arquitetura básica do Sistema MDX sobre uma rede de trabalho..........60
FIGURA 5.3 – Funcionamento de um autômato RS distribuído 64
FIGURA 5.4 – Processo de inicialização dos autômatos RS distribuídos 65
FIGURA 5.5 – Estrutura do novo núcleo de comunicação do MDX.............................66
FIGURA 5.6 – Arquitetura do antigo sistema MDX 67
FIGURA 5.7 – Arquitetura do novo sistema MDX 67
FIGURA 5.8 – Comunicação do *mouse* distribuído no novo sistema MDX 69
FIGURA 6.1 – Arquitetura do Controle do Três Olhos 72
FIGURA 6.2 – Reações do TÓ a um obstáculo (módulo FUGIR).................................72
FIGURA 6.3 – Limites das distâncias que o Três Olhos é sensível...............................73

Lista de Tabelas

TABELA 3.1 – Autômato e regras gerados para a *machine* sinope do *mouse* distribuído39
TABELA 3.2 – Autômato e regras gerados para a *machine* pan do *mouse* distribuído .. 39
TABELA 3.3 – Arquivo de informações distribuídas do *mouse* distribuído 40
TABELA 5.1 – NLT do novo núcleo MDX... 64

Lista de Abreviaturas

AID	Arquivo de Informações Distribuídas
API	*Application Program Interface*
ARSD	Autômato RS Distribuído
CRSD	Compilador RS Distribuído
IP	*Internet Protocol*
LAN	*Local Area Network*
MDX	Multimídia Distribuída em UNIX
NLT	*Name Local Table*
OC	*Object Code*
RPC	*Remote Procedure Call*
RS	Reativa Síncrona
RSD	RS Distribuída
RSP	RS Padrão (não-distribuído)
SPMD	*Simple Program Multiple Data*
SR	*Synchronizing Resources*
TCP	*Transmission Control Protocol*
UD	Unidades Distribuídas

Resumo

A Linguagem RS é destinada a programação de núcleos reativos centralizados. 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 saída. Sendo sua idéia inicial tratar apenas processos centralizados, não houve a preocupação com a distribuição.

Este trabalho tem como principal objetivo apresentar os aspectos introduzidos de uma nova versão para a Linguagem e para o Compilador RS, que possibilitam a execução de programas distribuídos. Além da possibilidade de execução de sistemas reativos distribuídos, foi acrescentado à Linguagem RS extensões já previstas na sua criação, como sinais inibidores, regras de exclusão mútua e concomitância, a possibilidade de disparo de mais de uma regra em um mesmo instante e a limpeza léxica do código fonte RS.

As modificações incorporadas nesta nova versão da linguagem, foram efetivadas através de um novo compilador, chamado de Compilador RS 5.0. O protótipo implementado oferece a geração de três formatos de código: o formato padrão da linguagem RS (os autômatos e as regras correspondentes), códigos na linguagem C para a simulação dos autômatos (tanto para programas distribuídos quanto não-distribuídos) e arquivos no formato portável OC, que é um formato de código objeto padrão para as linguagens reativas.

Para a distribuição e implementação da Linguagem RS foi necessária a criação de um novo núcleo de comunicação do MDX, que é responsável pela comunicação dos autômatos RSD. Este núcleo é dividido em três partes. A primeira trata da definição de um modelo formal com as mudanças necessárias para que a linguagem RS consiga trabalhar de forma distribuída, a segunda mostra o projeto do novo núcleo MDX e a terceira apresenta a implementação em C e MDX dos autômatos gerados pelo Compilador RS 5.0.

Por fim, exemplos de aplicação desta nova linguagem são apresentados, onde podem ser vistos a importância e o acréscimo proporcionado por este trabalho tanto à linguagem RS quanto à programação de sistemas reativos síncronos.

PALAVRAS-CHAVE: linguagem RS, núcleos reativos, compiladores, sistemas de tempo real, programação paralela e distribuída, MDX.

TITLE: "A Compiler for Distributed RS Language "

Abstract

The RS language is intended to the programming of centralized reactive kernels. Such kernels are responsible for the logic of a reactive system, manipulating the input signals, carrying through the reactions and generating the output signals. Being its initial idea to treat only centered processes, it did not have the concern with the distribution.

The main objective of this work is to describe the process of creation of a new version for the Language and Compiler RS, that make possible the execution of distributed programs. Beyond the possibility of execution distributed reactive systems, it was added to RS language foreseen extensions already in its creation, as inhibiting signals, rules of manual exclusion and concurrence, the possibility of detonation of more than a rule in one exactly instant and the lexical cleanness of the RS code source.

The modifications incorporated in this new version of the language, had been accomplished through a new compiler, called Compiler RS 5.0. The implemented archetype offers the generation of three formats of code: the standard format of RS language (the corresponding automatons and rules), codes in the language C for the simulation of the automatons and archives in OC portable format, that is a object format code standard for the reactive languages.

For the distribution and implementation of Language RS was necessary the creation of a new kernel of communication of the MDX, that is responsible for the communication of RSD automatons. It is divided in three parts. The first one deals with the definition of a formal model that defines the necessary changes so that RS language obtains to work of distributed form, the second shows the design of new MDX kernel and third presents the implementation in C and MDX of the automatons generated for Compiler RS 5.0.

Finally, examples of application of this new language are presented, where the importance and the proportionate upgrade for this work to RS language how to the programming of synchronous reactive systems can in such a way be seen.

KEYWORDS: RS language, reactive systems, reactive kernels, real-time systems, parallel and distributed programming, MDX.

1 Introdução

Sistemas reativos são sistemas que interagem dinamicamente com um ambiente externo, respondendo aos estímulos provenientes desse ambiente, alternando em dois tipos de períodos: espera o estímulo dos sinais e reage a eles. As linguagens reativas síncronas constituem ferramentas de programação de sistemas reativos. Estas linguagens adotam a hipótese de sincronismo, onde se presume que toda reação seja instantânea e, portanto, atômica. Isto corresponde a considerar sistemas ideais, que reagem instantaneamente a cada estímulo externo com uma transformação de estado interno e com uma emissão de sinais.

Os sistemas reativos são normalmente divididos em três camadas: uma camada de interface com o ambiente, um núcleo reativo e uma camada de manipulação de dados.

A linguagem Reativa Síncrona [TOS 93], que será referida como RS 4.0, destina-se à programação de núcleos reativos, que constituem a parte central e mais difícil de um sistema reativo. 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. Esta linguagem permite, a partir de um programa escrito em RS, gerar um autômato finito correspondente [MEN 97].

A linguagem RS constitui uma notação adequada para representar o comportamento de núcleos reativos, pois um programa é uma especificação quase direta das transformações internas e das emissões de sinais que devem acontecer para cada estímulo possível. A hipótese do sincronismo resulta em algumas vantagens para o programador tais como: reconciliar concorrência e determinismo, escrever programas simples e mais rigorosos e desassociar a lógica de um sistema das características dependentes de implementação, tais como tempos de reação.

A compilação de um programa RS é feita para um conjunto de tabelas que descrevem uma máquina de estados finita, similar à máquina de Mealy. A máquina inclui uma memória e ações para serem realizadas em tempo de execução. Como o resultado não é um arquivo executável nativo do processador, é necessário o uso de um interpretador em tempo de execução. A máquina de estados, sem interpretador, mais as camadas de interface e de manipulação de dados irão rodar em uma máquina hospedeira.

Sendo a idéia inicial da linguagem tratar apenas processos centralizados, não houve preocupação com o controle distribuído. Por conseqüência, os programas são sempre executados em um único processador e não existem facilidades que possibilitem a distribuição de um programa. A utilização de controles reativos distribuídos é uma realidade, sendo utilizado em diversos ambientes, como, por exemplo, no chão de fábrica, na automação doméstica e na robótica. Nesse ambientes, a reação de um componente pode depender do comportamento de *n* outros componentes.

O objetivo principal do trabalho é estender a linguagem RS em vários aspectos, de modo a torná-la mais adequada para a programação de sistemas de controle distribuídos. As extensões, como sinais inibidores e funções de exclusão mútua e concomitância, serão efetivadas através de um novo compilador, o Compilador RS 5.0, e deverão contemplar tanto aspectos sintáticos como semânticos. Com isso, pretende-se oferecer ao projetista uma

ferramenta de especificação de sistemas reativos distribuídos com elevado nível de abstração. Por fim, um ambiente de simulação para os autômatos RS distribuídos (ARSD), baseado na plataforma MDX, possibilitará a simulação dos programas RS 5.0.

A sintaxe atual da linguagem RS foi fortemente influenciada pela sintaxe de *termos* do Prolog. Isto simplificou em muito o reconhecimento dos programas, mas tornou um pouco deselegante o formato dos mesmos. Nesta implementação, a sintaxe de RS 5.0 foi aprimorada, em relação a RS 4.0, de cinco maneiras: (1) tornando a linguagem mais linear, eliminando o uso de colchetes para denotar listas; (2) generalizando o formato das condições de disparo, de modo a permitir o uso de mais de um sinal externo bem como o uso de sinais inibidores; (3) acrescentando uma nova declaração para especificar relações de exclusão mútua e de concomitância entre sinais externos; (4) simplificando a especificação das regras de exceção; e (5) tornando desnecessário o uso de declarações vazias.

Do ponto de vista semântico, as principais modificações da linguagem dizem respeito à implementação dos itens (2) e (3) referidos anteriormente. A possibilidade do uso de mais de um sinal externo no gatilho de uma regra de reação faz aumentar o detalhamento de possíveis reações de um programa (felizmente, não é aumentado o espaço de possíveis estados do programa, que é função dos sinais internos permanentes desse programa). O uso de sinais inibidores não aumenta o poder de representação da linguagem, mas facilita essa representação e não complica muito o processo de geração de código.

O Compilador RS 5.0, por ser desenvolvido em C, é auto-suficiente, não necessitando o uso de qualquer linguagem hospedeira. Além disso, sua versão final é multiplataforma, pois seu código é C-ANSI. Este compilador recebe como entrada um arquivo com um programa fonte RS (um programa distribuído, um programa centralizado ou um módulo). O programa implementa as análises léxica, sintática e semântica. Não havendo erros em nenhuma das análises, a próxima etapa é a geração de código.

Em relação ao código gerado, mantém-se a saída do Compilador RS original, isto é, a geração dos arquivos *autômato* e *regras*. Além deste, pode ser gerado um programa C (padrão ANSI) e também um arquivo com código no formato OC (Object Code), que está se tornando padrão como um formato comum para representação dos autômatos de estados finitos associados às linguagens reativas síncronas. O formato OC permite tradução eficiente em várias linguagens de programação, como por exemplo, C, ADA, EMC, etc., usando pós-processadores apropriados (esse pós-processador será recomendado como trabalho futuro). O código gerado pelo Compilador RS 5.0, no formato OC, terá características semelhantes às dos arquivos gerados em Esterel [BOU 91] e Lustre [HAL 91], duas linguagens reativas síncronas bem desenvolvidas.

Este trabalho define, também, um modelo formal com as mudanças necessárias para que a linguagem RS consiga trabalhar de forma distribuída, para realizar a tradução desse modelo para C e biblioteca MDX [PRE 98]. A partir dos autômatos distribuídos, gerados pelo Compilador RS 5.0, o sistema desenvolvido se encarrega da execução dos mesmos, de forma distribuída. O modelo tenta ser o mais genérico possível, não amarrando definições a linguagens ou a ambientes. Depois de definido o modelo formal da linguagem RS distribuída (RSD), o restante deste trabalho concentra-se na implementação do Compilador RS e do ambiente de execução de autômatos RS distribuídos. São apresentadas as principais características e vantagens da linguagem, os detalhes de implementação relevantes, as estruturas de dados mais importantes e os arquivos necessários no processo de compilação.

Esta dissertação tem como principais objetivos: a descrição de uma nova versão para a linguagem RS, a construção de um novo compilador e a criação de um novo núcleo de comunicação do MDX, que é responsável pela comunicação dos ARSDs no ambiente RS distribuído, tendo o restante dos seus capítulos organizados da maneira abaixo descrita.

O Capítulo 2 apresenta uma introdução aos sistemas reativos, com ênfase para os sistemas síncronos. Introduz os principais conceitos e particularidades da linguagem RS. Essencial para o entendimento dos capítulos posteriores, mostra a estrutura, a sintaxe e a semântica de seus principais comandos, bem como o resultado da compilação de um programa RS - o autômato RS - que será utilizado pelo Compilador RS 5.0 posteriormente.

O Capítulo 3 define a Linguagem RS 5.0. São apresentadas as mudanças na sintaxe da linguagem, em relação à versão anterior, como o uso de mais de um sinal disparador pelo ambiente externo, os sinais inibidores, as regras de exclusão mútua e concomitância, e a simplificação das regras de exceção. Este capítulo também apresenta as mudanças na semântica da linguagem, em relação às modificações sintáticas. Por fim, apresenta os passos para chegar à distribuição da Linguagem RS. São mostradas as modificações sintáticas, o código de entrada e de saída de programas distribuídos, o ambiente de execução, o formato de comunicação entre os autômatos RS distribuídos e, também, o protocolo RS.

O Capítulo 4 apresenta o Compilador RS 5.0. São apresentadas as análises léxica, sintática e semântica e é definido o formato dos arquivos a serem gerados pelo compilador. O formato é o *autômato-regras* que é usado para a geração dos demais. Os outros formatos são o código C, que pode ser utilizado para fazer a simulação do autômato, e o código no formato OC, o formato comum entre linguagens síncronas.

No Capítulo 5, apresenta-se o projeto do ambiente de execução RS distribuído. Primeiramente é apresentado o ambiente MDX, para programação paralela. São mostrados o modelo de programação, o núcleo de comunicação e os servidores. Após, é definido o novo núcleo de comunicação do MDX, que provêm suporte a comunicação entre os ARSD. Inicialmente é mostrado o que realmente é importante para a troca de mensagens entre ARSD, tendo por base o funcionamento dos próprios autômatos. Posteriormente, é vista a nova estrutura do núcleo MDX. O restante do capítulo apresenta todo o projeto da nova estrutura do núcleo, como o funcionamento do procedimento de estabelecimento de conexões, a troca de mensagens entre autômatos remotos e também é visto um exemplo de comunicação no sistema RSD, usando para isso um modelo do *mouse* distribuído.

No capítulo 6, são apresentados dois exemplos de aplicação da Linguagem RS 5.0. O primeiro exemplo é um pequeno veículo, que possui 3 sensores (por isso o nome "Três Olhos", ou simplesmente TÓ). O TÓ fica andando livremente até encontrar obstáculos. Se ele julga a distância perigosa, ele executa uma manobra defensiva. Caso encontrado algum objeto interessante de ser seguido, é acionado um módulo responsável pela perseguição. Quando não são encontrados objetos próximos, o TÓ fica vagueando até encontrar algum objeto. O segundo exemplo, mostra uma auto-estrada automatizada, onde os automóveis são totalmente automatizados, dispensando o uso de motorista. Os carros são controlados por uma central, que é auxiliada por sensores colocados em toda a extensão da pista, que informam a velocidade dos veículos, entre outras coisas.

Por fim, o capítulo 7 apresenta as conclusões desta dissertação, resumindo os pontos importantes, mostrando possíveis formas de continuar este projeto.

2 Os Sistemas Reativos e a Linguagem RS

Este capítulo apresenta uma introdução aos sistemas reativos e a linguagem RS. Inicialmente, são feitas considerações sobre os sistemas reativos, a sua definição e características. Da linguagem RS, serão definidas suas características, seus comandos principais e explicar-se-ão os autômatos por ela gerados.

2.1 Sistemas reativos

Sistemas reativos são sistemas que interagem com um ambiente externo, mantendo um relacionamento dinâmico com esse ambiente. Estes sistemas, introduzidos por Harel [HAR 85], devem responder a estímulos provenientes do ambiente externo, numa ordem desconhecida.

Harel propôs uma nova dicotomia para os sistemas computacionais. Além das clássicas (sistemas determinísticos x sistemas não-determinísticos, sistemas seqüenciais x sistemas concorrentes, sistemas discretos x sistemas contínuos, etc.), a nova dicotomia proposta foi: sistemas reativos x sistemas transformacionais.

Um sistema transformacional produz resultados a partir de um conjunto de dados, realizando transformações sobre os mesmos. Estes sistemas são considerados como funções das entradas disponíveis no momento de início da computação para a saída fornecida no término. Exemplos destes sistemas 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. O comportamento de um sistema reativo pode ser descrito basicamente por uma seqüência cíclica de três passos: espera estímulo, calcula respostas, emite sinais de saída.

Gérard Berry [BER 89] prefere excluir da categoria de reativos, os sistemas que interagem à sua própria velocidade (isto é, sem exigências do ambiente externo, em relação a tempo de resposta) com usuários ou outros programas. Estes sistemas são denominados interativos. Um sistema *timesharing* , por exemplo, é interativo, pois é ele que determina o ritmo das interações com o ambiente externo. 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 restritos 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.

Nesta dissertação, a denominação *sistema reativo* será usada no sentido mais restrito, conforme caracterizado por 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 [BER 92]:

- 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 (computações clássicas, normalmente implementadas por procedimentos externos escritos numa outra linguagem) 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 controladores. Os autômatos permitem obter excelentes desempenhos em tempo de execução e apresentam a vantagem de serem bem conhecidos matematicamente.

Linguagens de programação concorrentes são ferramentas mais elaboradas. Elas permitem o desenvolvimento hierárquico e modular de programas. Os mecanismos para controle de processos e comunicação são definidos ao nível da linguagem e, portanto, são portáteis. 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 [HEN 90]. 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 (por exemplo, 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 conseqüência da hipótese do sincronismo. Esta hipótese permite considerar a passagem do tempo como um evento sinalizado do exterior (e, como conseqüência, permite não considerar a sinalização de outros eventos durante o tratamento de um estímulo), o que resulta em simplicidade (e clareza) de semântica para os mecanismos de manipulação do tempo.

2.2 A linguagem RS

A linguagem RS, desenvolvida por Simão S. Toscani [TOS 93], destina-se à programação de núcleos reativos, que constituem a parte central e mais difícil de um sistema reativo. 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. Esta linguagem permite, a partir de um programa escrito em RS, gerar um autômato finito correspondente. Além do núcleo, uma aplicação reativa envolve a implementação de uma interface, responsável pela recepção dos estímulos e a condução das saídas, e de um conjunto de procedimentos, responsáveis pela manipulação dos dados requeridos pela aplicação.

A linguagem RS adota a hipótese de sincronismo entre os estímulos de entrada e os sinais de resposta, ou seja, o tempo só passa durante a atividade do ambiente externo. RS não é uma linguagem de propósitos gerais, nem tampouco auto-suficiente, tal como Esterel [BOU 91]. As camadas de interface e de manipulação de dados devem ser especificadas em alguma linguagem hospedeira, no caso do RS, o Prolog. Implementada em Prolog, RS compila seus programas para um conjunto de tabelas que descrevem uma máquina de estados finita, similar à máquina de Mealy [HOP 79]. O resultado não é um arquivo executável, requerendo assim o uso de um interpretador em tempo de execução.

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 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. Um programa pode ser organizado em diferentes módulos independentes, os quais atuam em paralelo sobre conjuntos privativos de variáveis e sinais. A organização modular é vantajosa, principalmente na programação de sistemas de maior porte.

Como exemplo de um sistema reativo, um controlador de metrô [CLA 83] 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.

2.2.1 Hipótese do sincronismo

A hipótese de sincronismo corresponde considerar sistemas ideais, que reagem instantaneamente a cada estímulo externo com uma transformação de estado interno e com uma emissão de sinais, portanto, uma reação atômica. Conforme já referido, ela eqüivale 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. Por outro lado, equivale a dizer que o ambiente externo permanece inalterado durante a reação.

A linguagem RS possui um 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 estímulo possível. A hipótese de sincronismo resulta em algumas vantagens para o programador como [TOS 93]: escrever programas simples e mais rigorosos e desassociar a lógica de um sistema das características dependentes de implementação, tais como tempos de reação [HAL 85].

Um programa RS deve ser visto como se as suas ações internas fossem executadas por um processador infinitamente rápido, em um tempo infinitesimal (o que importa é que o ambiente externo permaneça inalterado durante o tempo da reação). 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.

2.2.2 Sinais

Os sinais na linguagem RS são utilizados para comunicação com o exterior e para sincronização interna. 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*, e podem conter valores ou não. Se um sinal não contém valor ele é dito puro. A notação *s(V1)* indica que o sinal *s* contém o valor *V1*.

Portanto os sinais podem ser divididos em três conjuntos:

1. *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*.
2. *Sinais de saída*: são enviados ao ambiente externo para indicar os resultados das reações. São especificados com a declaração *output*.
3. *Sinais internos*: são usados para sincronização e comunicação interna de processos. Ainda são subdivididos em sinais temporários e permanentes:

- *Sinais internos 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.
- *Sinais internos permanentes*: ao contrário dos sinais temporários, eles 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*.

Pode-se ver um programa RS como um sistema que, ao ser estimulado por algum sinal, transforma-se internamente (muda de estado), emite sinais de saída e fica à espera de um novo sinal, tudo isto 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 sempre à espera de algum estímulo externo.

2.2.3 Variáveis

O programa RS trabalha com variáveis clássicas, compartilhadas a nível de módulo. São especificadas através da declaração *var*. Estas variáveis possuem apenas valores numéricos e seus nomes sempre devem iniciar por letra minúscula.

A linguagem RS permite que o programador trabalhe com variáveis tipo *record*, constituídas por vários campos de informação. Estas variáveis estruturadas são especificadas, juntamente com as variáveis aritméticas, na declaração *var*. Por exemplo:

var: [v, a(R), b(T)].

2.2.4 Caixas de regras

A linguagem RS permite agrupar regras logicamente relacionadas, em unidades sintáticas próprias para fins de ativação e desativação, para maior clareza dos programas. Estas unidades sintáticas são denominadas caixas de regras e os comandos para ativar e desativar essas caixas são:

activate (boxName) e *deactivate*

onde *boxName* é o nome de uma caixa de regras. O comando *deactivate* mão possui argumento, pois ele desativa a caixa de regras onde este comando é usado.

Para facilitar a programação, existe ainda o comando

exit_to(boxName)

o qual corresponde a desativar a caixa em que o comando é usado e ativar a caixa *boxName*. Ele é implementado pela seqüência: *activate(boxName)*, *deactivate*.

Inicialmente, todas as caixas estão inativas. Isto obriga que a ação de inicialização do programa inclua a ativação de pelo menos uma de suas caixas.

2.2.5 Exceções

O mecanismo para tratamento de exceções na linguagem RS é bastante poderoso e pode ser resumido da seguinte maneira: sempre que uma condição de exceção é sinalizada, o programa sofre uma mudança brusca de estado, como mostrado a seguir.

- 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.

Os comandos utilizados para tratar exceções são os seguintes:

- *on_exception*: usado para definir as condições e ações no caso de uma exceção.
- *raise*: usado para sinalizar uma condição de exceção durante uma reação.
- *reset*: usada para propagar uma condição de exceção.

2.3 Sintaxe

Um programa RS, cujo nome seja *programa* e que contenha *n* módulos, possui a seguinte forma [GIO 97]:

```
rs_prog programa
[ input: I,
  output: O,

        module m1:
        [   input: m1I,
            output: m1O,
            t_signal: m1T,
            p_signal: m1P,
            var: m1V,
            initially: m1A,
            on_exception m1E,
            m1R,
            .........
        ],
        .........................
        module mn:
        .....................
].
```

onde:

I é a lista de sinais de entrada;
O é a lista dos sinais de saída;
m_nI é a lista dos sinais de entrada do módulo n;
m_nO é a lista dos sinais de saída do módulo n;
m_nT é a lista dos sinais internos temporários do módulo n;
m_nP é a lista dos sinais internos permanentes do módulo n;
m_nV é a lista de variáveis do módulo n;
m_nA é a lista dos comandos de inicialização;
m_nE é a lista de regras de exceção;
m_nR é a lista das regras de reação (pode haver caixas de regra).

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

Uma regra de reação tem a forma: $F \Rightarrow A$, 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 seqüência

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

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

case
$[\ b_1 \rightarrow C_1,$
......,
$b_n \rightarrow C_n]$ $(n \geq 2)$

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

2.4 Algoritmo de execução

A seguir é apresentado o algoritmo para execução de um programa (a formalização deste algoritmo com sistemas de transição encontra-se em [TOS 93]). Cada reação é desencadeada por um estímulo vindo do exterior, o qual é materializado pela abertura e atribuição de valor a um sinal de entrada.

Após a ação de inicialização, a execução se processa repetindo o seguinte ciclo:

(1) Esperar pelo próximo sinal do exterior.

(2) Quando ocorrer um sinal externo, repetir as ações (a), (b) e (c), até que não hajam mais regras para serem executadas (depois ir para (3)):

(a) Identificar todas as regras que possam ser executadas, dados os sinais presentemente abertos.

(b) Fechar todos os sinais usados nos lados esquerdos das regras identificadas.

(c) Executar (em paralelo) as ações de todas as regras identificadas.

(3) Se (2) não tiver sinalizado algum evento de exceção, então voltar para (1). Caso contrário, executar as ações (a), (b) e (c), uma única vez:

(a) Fechar todos os sinais internos.

(b) Executar (em paralelo) as ações de todas as regras de exceção que tenham sinais (eventos) abertos.

(c) Fechar todos os sinais de exceção do programa.

A execução de uma reação se desenvolve, pois, em passos seqüenciais, com disparo paralelo de regras em cada passo (primeiro as regras globais, em um único passo, depois as regras locais, em 0 ou mais passos). A execução de uma regra tipo *case* consiste em executar a primeira opção que tenha condição verdadeira. Isto envolve a avaliação das expressões booleanas das opções, na ordem em que elas aparecem, até ser encontrada a primeira verdadeira.

Um programa só é aceito se ele satisfaz as três seguintes condições de ordem semântica: (1) jamais uma regra que emita sinal é repetida durante a execução de uma reação; (2) jamais as regras paralelas de um passo acessam variáveis de forma conflitante (acessos paralelos a uma variável só são aceitos se forem todos para leitura); (3) jamais um sinal é emitido mais de uma vez numa reação.

Conceitualmente, todas as regras de um passo são executadas simultaneamente, em tempo zero. Equivalentemente pode-se considerar uma execução escalonada, efetuada por um processador infinitamente veloz. Como o compilador só aceita programas que compartilham variáveis corretamente, então as regras paralelas de um passo podem ser escalonadas de maneira arbitrária, sem que haja mudança no resultado final da execução.

Essas três condições (ausência de *loops*, compartilhamento não conflitante de variáveis e uso correto de sinais) são suficientes para garantir execuções finitas e determinísticas para as várias reações. São estas condições que possibilitam a representação de um programa RS por um autômato finito estendido.

2.5 Semântica

A semântica lembra muito as redes de Petri [PET 77], pois a execução desenvolve-se em passos seqüenciais nos quais as regras com condição de disparo verdadeira são disparadas em paralelo. A condição de disparo de uma regra é verdadeira quando todos os seus sinais (que podem ser vistos como semáforos) estão abertos ou ligados. 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 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 em tais condições de disparo. Um fato importante a ser notado é que não existe indeterminismo na execução paralela de regras.

A execução de um programa processa-se, então, em uma 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 a desencadear uma nova reação.

Conforme 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 [PET 77], 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 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 caráter 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. Consequentemente, 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.

2.6 Autômato RS gerado

Em geral, as linguagem reativas síncronas geram um código objeto que é um autômato finito. A linguagem RS também apresenta a mesma característica. O comportamento de um programa RS pode ser representado por um autômato finito [MEN 97]. Na realidade, este autômato que representado um programa RS é um autômato estendido, similar a máquina de

Mealy, o qual inclui uma memória e ações para serem realizadas em tempo de execução. Convém observar que os autômatos que representam sistemas reativos normalmente não apresentam estados finais, pois os programas reativos, em geral, são cíclicos.

O código gerado pelo compilador RS é composto por duas partes: o autômato e as regras ligadas a esse autômato. As duas partes são interdependentes e devem ser estudas em conjunto.

Apresentamos a seguir um programa simples RS, com um único módulo, que representa o funcionamento de uma lâmpada. Os sinais de entrada são "*on*" para ligar a lâmpada se estiver desligada e "*off*" para desligar, quando ela estiver ligada. Quando a lâmpada estiver ligada e o sinal de entrada for "*on*", é emitido um sinal de alerta ("*ring*"). O mesmo acontece para quando estiver desligada e receber o sinal "*off*". O programa RS referente está descrito abaixo:

```
module lamp:
[ input :[on, off],
  output:[ligada, desligada, ring],
  t_signal:[ ],
  p_signal:[a,b],
  var:[ ],
  initially:[up(a),activate(rules)],
  on_exception:[ ],
  on#[a]===>[emit(ligada),up(b)],
  off#[a]===>[emit(ring),up(a)],
  on#[b]===>[emit(ring),up(b)],
  off#[b]===>[emit(desligada),up(a)]
].
```

Segue o código gerado para o programa mostrado acima, que representa o funcionamento da lâmpada:

```
AUTOMATON
init -    [1, *, go_to(1)]
  1  on   [2, *, go_to(2)]
  1  off  [3, *, go_to(1)]
  2  on   [4, *, go_to(2)]
  2  off  [5, *, go_to(1)]

RULES
Module lamp:
  1.  [ ] ===> [ ]
  2.  [ ] ===> [emit(ligada)]
  3.  [ ] ===> [emit(ring)]
  4.  [ ] ===> [emit(ring)]
  5.  [ ] ===> [emit(desligada)]
```

A palavra *AUTOMATON* inicia a definição de um autômato, que corresponde como à descrição de uma máquina de estados. O número à esquerda representa um estado do autômato, sendo que *init* é o estado inicial. Cada linha possui três componentes identificados *n*, *s*, *a*, onde *n* é um número de estado, *s* é um nome de sinal externo e *a* é a ação a ser executada quando, no estado *n*, o autômato é estimulado pelo sinal *s*. Os asteriscos separam regras que podem ser executadas em paralelo. A Figura 2.1 sintetiza o formato de uma linha do autômato.

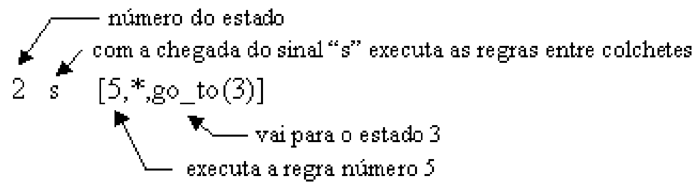

FIGURA 2.1 – Linha de um autômato gerado por RS

O máquina de estados correspondente ao autômato é representado na Figura 2.2.

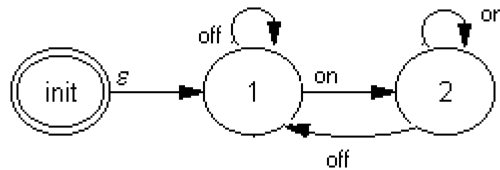

FIGURA 2.2 – Máquina de estados correspondente ao programa RS

A segunda parte inicia com a palavra "RULES". Nesta parte, são descritas as ações a serem executadas por um autômato. O formato geral de uma regra é a seguinte:

n. [condição] ===> [ação]

Neste caso, *n* é o número de identificação da regra, referenciado no autômato. A "condição", pode ser vazia ou não. Ela tem que ser verdadeira para a regra ser executada. A "ação", também pode ser vazia ou não. E no caso de não ser, pode conter vários comandos, todos separados por vírgulas. No exemplo abaixo, toda vez que uma das regras do autômato "chamar" a regra de execução 4, o sistema reativo emitirá o sinal "*ring*".

4. [ ] ===> [emit(ring)]

2.7 Considerações sobre a utilização de autômatos

Uma das grandes vantagens da utilização de linguagens reativas síncronas é a confecção de programas sem se preocupar com os tempos de resposta ou interrupções prioritárias durante o desenvolvimento de uma reação, resultando em uma programação bem mais simples. Estas complicações são abstraídas devido a hipótese de sincronismo.

Após a compilação do programa, é gerado o código objeto, cuja execução vai requerer um tempo, evidentemente. Só que o código gerado pelas linguagens síncronas é um autômato finito determinístico e eficiente. Com isto, o tempo de resposta pode ser calculado anteriormente a sua execução, a partir do código gerado para o autômato. Então, o que se faz é: especificar o comportamento reativo usando uma lógica simples, gerar o código (autômato) e verificar se os tempos de resposta são satisfatórios para a aplicação. Caso os tempos de resposta sejam satisfatórios, está pronto o programa. Se não forem suficientes é porque ou o

sistema não pode ser controlado por um computador, ou é necessário uma máquina mais rápida para tal, pois o código gerado já é extremamente eficiente [MAT 99].

A eficiência do código gerado é devida a simplicidade das linguagens síncronas [TOS 96]. Esta simplicidade permite que um programa reativo seja compilado para um autômato seqüencial, mesmo sendo o programa concorrente. Sendo assim, pode-se afirmar que a grande vantagem da linguagem síncronas é elas poderem combinar a facilidade da programação das linguagens concorrentes com a eficiência dos autômatos.

2.8 Conclusões sobre a Linguagem RS

A hipótese do sincronismo permite considerar a passagem do tempo como um evento sinalizado do exterior (como conseqüência, permite não considerar a sinalização de outros eventos durante o tratamento de um estímulo), o que resulta em clareza de semântica, e a ausência de condições de corrida permite escalonar os processos paralelos de forma arbitrária. Estas suposições fornecem as condições necessárias para a geração de códigos eficientes em tempo de execução. Como qualquer seqüenciação dos processos paralelos leva ao mesmo resultado, isto reduz em muito o espaço de estados do programa, viabilizando a pesquisa exaustiva desse espaço.

O autômato de um programa RS é obtido através da varredura de todos os possíveis *estados de espera* do programa. Cada estado de espera é representado por um estado do autômato (nessa representação, cada transição do autômato define uma reação do programa para um estímulo do exterior). A obtenção do autômato de um programa através da exploração exaustiva do espaço de estados é um processo de compilação aceitável na prática. Convém observar que 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.

Pode-se questionar o realismo da hipótese do sincronismo, pois ela assume que a máquina reage instantaneamente aos estímulos do exterior. Contudo, 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.

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. Isto é, programação mais simples, sem a necessidade de se preocupar com a chegada de outros sinais durante a reação a um estímulo. Porém, as linguagens síncronas não resolvem todos os problemas. Elas não permitem, por exemplo, a criação dinâmica (ou recursiva) de processos. Na verdade, elas não passam de ferramentas especializadas para programação muito eficiente de núcleos reativos.

3 A Linguagem RS 5.0

Este trabalho tem por objetivo estender a linguagem RS em vários aspectos inexistentes na versão original, de modo a torná-la mais adequada para a programação de sistemas de controle distribuídos. A idéia é poder utilizar a linguagem no desenvolvimento de aplicações de maior porte. As extensões serão efetivadas através de um novo compilador RS 5.0, escrito em C, que suportará alterações, tanto em aspectos sintáticos quanto semânticos.

3.1 Mudanças na sintaxe de RS 5.0

A sintaxe atual da linguagem RS foi fortemente influenciada pela sintaxe de *termos* do Prolog. Isto simplificou em muito o reconhecimento dos programas, mas tornou um pouco deselegante o formato dos mesmos. Nesta implementação, a sintaxe de RS 5.0 foi aprimorada em vários aspectos, tais como: (1) a linguagem foi tornada mais linear, eliminando o uso de colchetes para denotar listas; (2) o formato das condições de disparo foi ampliado, de modo a permitir o uso de mais de um sinal externo e o uso de sinais inibidores; foi acrescentada uma nova declaração para especificar relações de exclusão mútua e de concomitância entre sinais externos; foi simplificada a especificação das regras de exceção.

3.1.1 Tornando a linguagem RS mais linear

Na versão 4.0 da linguagem RS, o uso de colchetes era muito evidente. O seu uso era necessário para a denotação de todas as listas do programa. Tanto as listas de declarações (*input, output, var, ...*), quanto listas de regras de execução. Isso acarretava em certo trabalho ao programador, pois como a linguagem RS permite o uso de uma lista dentro de outra lista (por exemplo, dentro de uma lista *case*, poderemos ter *n* listas de condições e regras de execução), deve-se ter muito cuidado com a abertura e o fechamento das mesmas.

Para resolver este problema, foi eliminado o uso de colchetes para a denotação de listas. Isso foi facilitado porque todas as listas (tanto de declarações, quanto de regras de execução) sempre tem seu início depois do *token* ":" (dois pontos) ou do *token* "==>" (seta à direita). Neste caso, o próximo *token* sempre será o "[" (abre colchetes), significando o início de tais listas. E o seu final, é sempre delimitado pela seqüência de *tokens* "]" (fecha colchetes) e "," (vírgula).

A solução adotada, foi retirar o uso do *token* "[" para a inicialização de uma lista, pois se sabe onde será o seu início. E em vez de finalizar a mesma, com os *tokens* "]" e ",", faz-se a finalização apenas com o *token* ";" (ponto e vírgula). Assim sendo, o código da linguagem RS 5.0 torna-se mais leve que o código similar da versão de RS 4.0.

Outra simplificação foi tornar desnecessário o uso de listas vazias. Na versão RS 4.0, mesmo que as listas de declarações (*t_signal, p_signal, var, initially* e *on_exception*) não

tivessem um único elemento, elas deveriam estar presentes no código fonte RS. Como solução, na linguagem RS 5.0, parte-se do pressuposto que se uma declaração não está presente no código fonte, ela automaticamente é vazia. Portanto a sua declaração não é necessária.

Para exemplificar essas modificações sintáticas (eliminação do uso de colchetes para delimitação de listas e retirada de declarações vazias), será usado um exemplo simples já apresentado anteriormente. É o programa RS que mostra o funcionamento de uma lâmpada, com 2 interruptores (um para ligá-la e outro para desligá-la). Estes sinais, portanto, são as entradas desse sistema. Como saída, o programa mostra se a lâmpada está ligada ou desligada. O sistema também pode dar um aviso de erro, quando o usuário entra com o sinal *on*, estando a lâmpada ligada; ou quando o usuário entra com o sinal *off*, estando a lâmpada desligada.

No código da Figura 3.1, está o programa RS descrito para a versão 4.0 da linguagem RS. Note o uso de colchetes e de listas vazias, na sua codificação.

```
module lamp:
[ input :[on, off],
  output:[lig, deslig, ring],
  t_signal:[ ],
  p_signal:[a,b],
  var:[ ],
  initially:[up(a), activate(rules)],
  on_exception:[ ],
  on#[a]===>[emit(lig), up(b)],
  off#[a]===>[emit(ring), up(a)],
  on#[b]===>[emit(ring), up(b)],
  off#[b]===>[emit(deslig), up(a)]
].
```

FIGURA 3.1 – Código da lâmpada em RS 4.0

Na Figura 3.2, está o programa RS reescrito para a versão 5.0. A eliminação dos colchetes supérfluos e das listas vazias resultou em um código mais leve e elegante.

```
module lamp:
  input    : on, off;
  output  : lig, deslig, ring;
  p_signal : a, b;
  initially : up(a);
  on+[a]  ==> emit(lig), up(b);
  off+[a] ==> emit(ring), up(a);
  on+[b]  ==> emit(ring), up(b);
  off+[b] ==> emit(deslig), up(a);
end module.
```

FIGURA 3.2 – Código da lâmpada em RS 5.0

3.1.2 O uso de mais de um sinal disparador do ambiente externo

Até a versão 4.0 da linguagem, um programa RS era estimulado por um único sinal do ambiente externo, de cada vez. Tendo por inspiração a linguagem Esterel, onde um programa pode receber mais de um sinal externo do ambiente ao mesmo tempo, resolveu-se adotar essa política na versão 5.0 da linguagem RS.

A partir desta versão da linguagem, o ambiente externo pode estimular o programa com múltiplos sinais. Convém observar que o número máximo de estados do sistema permanece inalterado, em relação ao mesmo sistema programado na versão 4.0, onde apenas um sinal externo pode estimular o sistema de cada vez. Isso porque o número máximo de estados é função do número de sinais internos permanentes do programa. Se o programa tem *n* sinais internos permanentes, o número máximo de estados dp autômato correspondente é 2^n.

Sintaticamente, não haverá mudanças nas listas de declarações. Porém, todos os sinais declarados na seção *input* do programa RS 5.0 poderão chegar do ambiente externo, ao mesmo instante. Para restringir as possibilidades de chegadas de sinais existe a regra de exclusão mútua (ver seção 3.1.6). Se não há restrição em relação às possíveis chegadas de sinais e o programa declarou *n* sinais de entrada, então o número total de combinações possíveis é 2^n.

3.1.3 Os sinais inibidores

Com a inclusão da possibilidade de poder receber mais de um sinal disparador do ambiente externo a cada reação, tornou-se conveniente adicionar à linguagem RS, os sinais inibidores. Estes sinais, tem como função exatamente o contrário dos sinais disparadores: como os sinais disparadores são necessários para o disparo de uma regra de reação, os sinais inibidores deverão estar ausentes para que esta regra seja disparada. Vale ressaltar que cada sinal pode assumir a condição de sinal disparador ou inibidor. Além disso, tanto os sinais externos quanto internos (temporários ou permanentes) podem ser usados como inibidores.

O uso de sinais inibidores amplia a abrangência da linguagem RS, pois eles permitem um melhor controle do sistema a ser desenvolvido.

3.1.4 O novo formato das condições de disparo das regras de execução

Nas versões anteriores da linguagem RS, cada condição de disparo de uma regra de reação, só podia ser formada por um único sinal externo e por sinais internos do programa (temporários ou permanentes). Conforme visto nas seções anteriores, foi introduzida na linguagem a possibilidade de usar mais de um sinal externo disparador em cada regra, além do uso de sinais inibidores, tanto internos, quanto externos.

Para formalizar isso sintaticamente, foi generalizado o formato das condições de disparo. Para exemplificar, vamos considerar a seguinte regra de disparo:

```
s1, s2+[a,b]#{s3, s4+[c,d]} ==> up(b);
```

onde nesta regra tem-se que *s1* e *s2* são sinais externos disparadores, *a* e *b* são sinais internos disparadores, *s3* e *s4* são sinais externos inibidores, *c* e *d* são sinais internos inibidores.

Para o comando *up(b)* ser executado, é necessário que os sinais *s1* e *s2* tenham sido disparados pelo ambiente externo e, ainda, o ambiente não tenha disparado os sinais *s3* e *s4*, pois nesta regra, eles tem a função inibidora. Além disso, os sinais internos *a* e *b* deverão estar

ligados, assim como os sinais internos *c* e *d* devem estar desligados. Se todas as condições acima forem satisfeitas, o comando *up(b)* será executado.

3.1.5 As declarações de Exclusão Mútua e Concomitância

Juntamente com os sinais inibidores, outra novidade da linguagem RS 5.0, é a declaração que especifica relações de exclusão mútua e de concomitância entre sinais externos. Para isso foi introduzida a nova declaração *relation* na linguagem.

Se esta declaração apresentar, entre dois sinais externos *a* e *b*, o caracter #, (*a* # *b*), significa que os sinais *a* e *b* são mutuamente exclusivos, ou seja, eles não podem ser disparados pelo ambiente em um mesmo instante. É uma forma do programador prever algumas possibilidades de erro, como no caso da entrada de dois sinais que poderiam disparar várias regras em um mesmo passo que não deveriam ser executadas, pois causariam uma inconsistência no sistema. Mas, sobretudo, é uma forma de reduzir o espaço de possibilidades a ser examinado, na geração do autômato.

O exemplo da Figura 3.3, que mostra o funcionamento de uma roleta de cinema. Os sinais provenientes do ambiente externo são *open* (para abrir a contagem de pessoas que passarão pela roleta), *close* (que fecha a contagem) e *people* (que indica que passou uma pessoa pela roleta). Note que os sinais *open* e *close* são conflitantes. Por isso, o programador se preocupou em especificar, através da primitiva *relation* que estes sinais são mutuamente exclusivos.

```
module roleta:
  input : open, close, people;
  output : openRoleta, closeRoleta, people(X), ring;
  relation : open # close;
  p_signal : a, b;
  var : peopleCount;
  initially : up(a), emit(closeRoleta);
  open+[a] ==> peopleCount:=0, emit(openRoleta), up(b);
  close+[a] ==> emit(closeRoleta), up(a);
  people+[a] ==> emit(ring), up(a);
  open+[b] ==> emit(ring), up(b);
  close+[b] ==> emit(people(peopleCount)), emit(closeRoleta), up(a);
  people+[b] ==> peopleCount:=peopleCount+1, up(b);
end module.
```

FIGURA 3.3 – Código de uma roleta de cinema em RS 5.0

Se a declaração *relation* apresentar o caracter >, (*a* > *b*), significa que, toda vez que o sinal *a* estiver presente, automaticamente o sinal *b* também estará. Isso implica que o programador está prevendo que um sinal pode, implicitamente, implicar em outro sinal externo, sem que o ambiente externo o tenha disparado. Como exemplo, pode-se citar um caso onde tenha que se controlar o tempo, e como sinais de entrada tenhamos *S* (segundo) e *mS* (milisegundo). Se relacionarmos *S* e *mS* como sinais concomitantes, nesta ordem, sempre que o sinal *S* ocorrer, automaticamente o sinal *mS* também ocorrerá. A regra seria declarada da seguinte forma:

relation : S > mS;

Tanto na exclusão mútua, quanto na concomitância, pode haver mais de 2 sinais externos na mesma declaração (i.e., *relation: a # b # c # d*, ou *relation: x > y > z*), mas na mesma declaração não se pode especificar as duas relações (i.e., *relation a # b < c*).

A relação abaixo especifica que os sinais externos *a, b, c, d* são mutuamente exclusivos, isto é, não podem ocorrer em nenhum passo de reação ao mesmo instante.

relation: *a # b # c # d*

Na declaração de relação a seguir, está sendo declarado que toda vez que ocorrer o sinal *x*, os sinais *y* e *z* também ocorrerão. Se apenas o sinal *y* for disparado, automaticamente o sinal *z* também será disparado. O sinal *z* ocorrerá sempre que ocorrer qualquer um dos sinais anteriormente a ele declarados na primitiva *relation*.

relation: *x > y > z*

3.1.6 A simplificação da especificação das regras de exceção

Como visto anteriormente, toda vez que uma exceção ocorre, o programa RS sofre uma alteração brusca no seu funcionamento, pois todos os módulos sofrem um *reset* (desligam-se todos os sinais internos e desativam-se todas as caixas de regras). Então, cada módulo envolvido passa para um novo estado, este sendo definido pela regra de exceção correspondente à condição que foi sinalizada. Para fazer essa sinalização de exceção a todos os módulos (para propagar essa condição de exceção), o comando *reset* era utilizado, na linguagem RS 4.0.

A partir da versão RS 5.0, o programador não terá que se preocupar com essa tarefa. Agora, isso passa a ser uma função do próprio compilador, pois ao detectar uma condição de exceção, automaticamente todos os sinais internos e todas as caixas de regras dos módulos envolvidos nesta exceção serão desligados.

Resumindo, foi apenas realizado uma simplificação nas regras de reação, retirando a primitiva *reset*, responsável pela propagação de um evento por todos os módulos. Os trechos de código das figuras 3.4 e 3.5 ilustra a diferença.

```
on_exception:
[        click ==> [activate(box)], reset [m1#C1]
],
```

FIGURA 3.4 – Trecho de código em RS 4.0

```
on_exception:
        click ==> activate(box);
end on_exception;
```

FIGURA 3.5 – Trecho de código em RS 5.0

Apesar de parecer uma mudança simples para a programação de sistemas reativos síncronos, essa simplificação despreocupa o programador em verificar quais os módulos e quais as caixas de regras que terão que ser desligadas. A partir do compilador RS 5.0, o

programador apenas prevê os casos de exceção, deixando com o compilador a tarefa de desativar sinais internos e caixas de regras envolvidas.

3.2 Mudanças na Semântica de RS 5.0

Do ponto de vista semântico, as principais modificações da linguagem dizem respeito à implementação dos itens referidos anteriormente. A possibilidade do uso de mais de um sinal externo no gatilho de uma regra de reação faz aumentar o detalhamento de possíveis reações de um programa (felizmente, não é aumentado o espaço de possíveis estados do programa, que é função dos sinais internos permanentes desse programa). O uso de sinais inibidores não aumenta o poder de representação da linguagem, mas facilita essa representação e não complica muito o processo de geração de código.

3.2.1 O disparo de mais de um sinal externo no gatilho

Com este novo compilador, a linguagem RS passa a permitir que um programa possa ser estimulado por mais de um sinal de entrada. Essa possibilidade foi acrescentada à linguagem devido a algumas situações que devem ser tratadas após a entrada de, no mínimo, dois sinais de entrada. Vamos exemplificar com um mecanismo de prevenção a incêndio. Para uma válvula de emergência ser aberta, seria necessário que viesse um sinal de um detector de fumaça e outro sinal de um detector de calor.

Na linguagem RS 4.0, esses sinais não poderiam ser estimular o programa ao mesmo tempo, portanto, deveria ser tratado a possibilidade de vir primeiro o sinal do detector de calor e depois o do detector de fumaça do ambiente externo, assim como de o sinal do detector de fumaça vir antes do detector de calor. Para solucionar este problema, a linguagem RS 5.0 passou a permitir a recepção de mais de um sinal do ambiente externo ao mesmo tempo.

3.2.2 O uso de sinais inibidores

O uso de sinais inibidores facilita a programação em RS, pois deixa de ser necessário utilizar variáveis temporárias para evitar o disparo de regras em momentos não propícios.

Em relação à geração de código, a inclusão de sinais inibidores não dificultou a criação da rede RS e a geração de autômatos. Apenas criou-se mais uma verificação, que é responsável por testar os sinais inibidores da regra de execução.

3.2.3 Problemas encontrados com o uso de vários sinais externos no gatilho

A possibilidade de se ter mais de um sinal como estímulo do ambiente externo aumenta a probabilidade de ocorrerem situações de inconsistência no programa, isto é, de serem disparadas regras que acessam variáveis globais de forma conflitante, ou que emitem sinais “simultâneos” sem que haja função para composição dos mesmos. A responsabilidade pelo tratamento dessas situações é do próprio programador, que pode, por exemplo, prever esse acontecimento e declarar os sinais externos causadores desse problema, na seção *relation*, como sinais mutuamente exclusivos. A situação se complica quando dois sinais devem ser mutuamente exclusivos apenas para alguns estados do sistema, mas não durante toda a execução do programa. Nesse caso, a solução deve ser encontrada através do uso de sinais inibidores.

Felizmente o compilador detecta e alerta o programador sobre a existência dessas situações de conflito, as quais são inaceitáveis para um programa reativo. Ao encontrar uma situação como essa, um erro de compilação é reportado ao usuário e o processo de compilação é encerrado. Portanto, nenhum arquivo de saída é gerado.

As várias situações possíveis serão exemplificadas através programa “mouse distribuído”, que será tornado ainda mais irreal para caracterizar bem as diversas situações. Na sua versão correta, é suposto que o programa nunca vai ser estimulado simultaneamente pelos sinais *click* e *tick*. Nesse caso, o código gerado para o autômato RS é tal que no estado 1 o sistema espera os sinais *click* e *tick*. Respectivamente, eles atualizam o estado atual do sistema para 2 e 1. A figura 3.6 mostra o trecho de código correspondente.

Se, por algum erro, no estado 1, o ambiente estimular o autômato com os sinais *click* e *tick*, qual seria o novo estado do sistema? O estado 2 ou o estado 1?

```
1  click [2, *, go_to(2)]
1  tick  [3, *, go_to(1)]
```

FIGURA 3.6 – Regras a serem disparadas em um mesmo estado

Deve ficar claro que o código acima só é gerado quando os sinais *click* e *tick* são declarados como mutuamente exclusivos. Portanto, se a interface com o ambiente externo forçar (como deve ser) o cumprimento dessa condição, a situação que estamos a imaginar jamais ocorrerá.

Se os sinais *click* e *tick* não são declarados como mutuamente exclusivos, o compilador vai indicar erro durante a compilação conforme é ilustrado a seguir.

Para evitar esse problema, provocado pelo acréscimo à linguagem RS 5.0 da possibilidade de disparo de mais de um sinal ao mesmo tempo pelo ambiente externo, deve ser adotado o seguinte procedimento:

1. O programador se preocupará em verificar quais as regras que podem levar a essa situação de inconsistência, e colocará os sinais que podem causar problemas em condição de exclusão mútua (e o ambiente externo não poderá disparar estes sinais ao mesmo instante);

2. Se os sinais não forem sempre mutuamente exclusivos, o programador tem a possibilidade de colocar os sinais que não podem ocorrer ao mesmo instante como inibidores um do outro, evitando que regras conflitantes possam ser disparadas ao mesmo tempo. Neste caso surge a possibilidade de situações de *deadlock*, as quais, felizmente também são detectadas pelo compilador.

3. Se o programador não tomar nenhuma providência em relação ao tratamento do problema, o próprio compilador fará as verificações e não aceitará o programa, se este contiver ações conflitantes.

Na sua pesquisa exaustiva, onde percorre todos os estados possíveis do programa, o compilador detecta qualquer situação de inconsistência e gera avisos de erro para o usuário. Sempre que um erro é encontrado, a compilação é abortada evitando que seja gerado algum arquivo de saída.

No exemplo do mouse distribuído, caso o programador não tivesse usado a declaração "*relation click # tick*", que declara *click* e *tick* como sinais mutuamente exclusivos e nem tivesse a preocupação de usar esses sinais como inibidores, seria gerada uma mensagem para o usuário.

Caso o programador optasse pela solução através de sinais inibidores o código gerado seria o mostrado na figura 3.7.

```
1  click#tick [2, *, go_to(2)]
1  tick#click  [3, *, go_to(1)]
1  click, tick  [4, *, go_to(1)]
```

FIGURA 3.7 – Regras a serem disparadas em um mesmo estado

Quando chega do ambiente externo apenas o sinal *click*, a primeira regra é executada. Quando o estímulo é apenas o sinal *tick*, a segunda regra é disparada. No caso do estímulo conter ambos os sinais *click* e *tick*, somente a terceira regra é executada, pois as duas regras anteriores possuem esses sinais como inibidores.

3.3 A Distribuição da Linguagem RS

Esta seção visa definir um modelo que permita executar os programas compilados em RS em uma ou mais máquinas [LI 97]. Como vimos no capítulo anterior, o compilador RS 4.0 gera um único autômato, não permitindo que um programa possa ser executado parte em uma máquina e parte em outra [GIO 98].

Nas seções subseqüentes, serão apresentados os passos que foram necessários para que a distribuição se tornasse possível. Em primeiro lugar, é apresentada a inclusão de um novo comando, nomeado *machine*, o qual permite que o programador divida um programa em diversas unidades funcionais e especifique onde cada unidade irá rodar. Em segundo lugar, resolvendo o problema inserido com a descentralização de RS, definimos como deve ser feita a comunicação entre autômatos, através do protocolo RS.

3.3.1 Modificações sintáticas

Como a linguagem RS foi desenvolvida com o intuito de seus programas rodarem em apenas um processador, nenhum comando, primitiva ou meio de comunicação entre processos foi previsto.

Do ponto de vista sintático, devemos acrescentar um ou mais comandos à sintaxe da linguagem RS que permitam a especificação de mais de um autômato e a alocação desses autômatos à diferentes processadores.

Duas abordagens podem ser utilizadas para gerenciar os diversos autômatos. A primeira, usada pelo MDX [PRE 98], exige que se tenha um arquivo separado para cada processo a ser distribuído. Nesse caso, tem-se um arquivo para cada máquina. A segunda permite que se especifique, em um mesmo código fonte, os diversos processos a serem distribuídos, transferindo para o compilador o trabalho de separá-los.

Embora as vantagens da segunda abordagem, utilizada por SR [AND 93], sejam evidentes[1], certos cuidados devem ser tomados para que ela possa ser usada. Estando diversas unidades funcionais no mesmo código fonte, deve-se ter mecanismos que impeçam que variáveis sejam compartilhadas entre elas de uma forma desordenada. Tais unidades funcionais devem ser bem delimitadas para permitir que o compilador identifique o início e o fim destas e consiga, assim, gerar um autômato para cada uma delas.

Para a linguagem RS esta abordagem é natural. A primeira restrição, o uso disciplinado de variáveis entre as partes distribuídas, pode ser facilmente resolvida. Sendo as variáveis RS compartilhadas a nível de módulo, basta exigirmos que as unidades distribuídas (UDs) tenham uma hierarquia igual ou maior a de um módulo. Já a segunda restrição, impondo que as UDs sejam bem delimitadas, faz com que seja necessária a inclusão de um novo comando RS. Para isso é utilizado o comando *machine*, o qual permite delimitar as UDs e gerenciá-las.

3.3.2 O comando *machine*

Este comando introduz uma nova hierarquia na linguagem RS. Originalmente, RS continha três níveis de estruturação. Um programa era composto por n módulos ($n \geq 1$), formados por k caixas de regras ($k \geq 0$) que, por sua vez, podiam ser compostas por m comandos ou regras de execução ($m \geq 0$).

Com a inclusão de *machine*, insere-se uma nova hierarquia que, propositalmente, iguala-se a de um programa RS, ou seja, dentro de um comando *machine* pode-se inserir todos os antigos níveis de estruturação. Como veremos adiante, a sintaxe aditada não só resolve as restrições de tratamento disciplinado de variáveis e de delimitação de UDs, como também

---

[1] A partir do momento em que as diversas unidades distribuídas podem ser vistas em um mesmo código, a programação de um sistema distribuído torna-se mais simples e mais intuitiva, como provado pela linguagem SR que é considerada uma das mais fáceis linguagens de programação do gênero.

resolve o problema identificado no início deste capítulo – possibilitar que a linguagem defina mais do que um autômato.

Vejamos, então, como fica a estrutura de um programa RS com a nova primitiva:

1. ***rsd_prog*** **nome_programa**: define o início de um programa RS distribuído. Pode conter uma ou mais primitivas *machine*.
2. ***machine*** **nome_máquina**: torna-se agora o primeiro nível na hierarquia. Corresponde ao antigo *rs_prog*. Tal primitiva deve estar dentro de um programa RS distribuído (*rsd_prog*) e define que o componente a seguir irá rodar na máquina *nome_máquina*. Pode ser formada por um único módulo ou ser subdividida em módulos paralelos. Em qualquer caso, um único autômato será gerado.
3. ***module*** **nome_módulo**: corresponde ao módulo da linguagem RS 4.0. Pode ser formado por diversas caixas de regras ou comandos.
4. ***caixas de Regras***.
5. ***comandos RS*** **(regras de execução)**.

Outra pequena mudança sintática foi efetuada para possibilitar uma programação mais clara. Após o cabeçalho *rsd_prog,* segue a definição da interface externa. A declaração *external interface* possibilita que se identifique os sinais externos de entrada e saída. Com isso, um programa RS distribuído sempre começará com o seguinte bloco de código:

```
rsd_prog Nome:
external interface:
[ input: [lista de sinais externos de entrada]
  output: [lista de sinais externos de saída]
].
```

FIGURA 3.8 – Cabeçalho de um programa RS distribuído

Deve-se observar que esta declaração da interface externa visa apenas a deixar mais claro o programa. As informações contidas na declaração são redundantes e poderiam ser obtidas diretamente do compilador, usando-se as seguintes regras: (1) todo sinal de entrada de um comando *machine* que não seja sinal de saída de outro deve, obrigatoriamente, ser provido do exterior; (2) todo sinal de saída de um comando *machine* que não seja sinal de entrada de outro deve, obrigatoriamente, ser enviado ao ambiente externo.

A sintaxe completa do comando *machine* é a mesma de *rs_prog* da versão 4.0 e será mostrada a seguir:

machine ***nome_máquina*****:**
[input: I,
output: O,
modulo$_1$,
  ...... ,
modulo$_n$
].

FIGURA 3.9 – Modelo de declaração de uma *machine* em um código RS

Um programa RSD poderá conter diversas declarações *machine* para um mesmo *nome_máquina*. Cada declaração *machine* irá originar um autômato independente para a máquina *nome_máquina*.

Para que se tenha uma idéia mais clara da utilização da nova sintaxe, apresentamos, a seguir, um programa RS que verifica se o botão de um *mouse* foi pressionado com um *click* duplo ou simples. Ele possui apenas dois sinais de entrada: *tick*, que corresponde a um impulso de relógio, e *click*, que corresponde ao pressionamento do botão do *mouse*. Possui, também, apenas dois sinais de saída: *single* e *double*.

O programa será apresentado em duas versões. A primeira é um programa RS 4.0 (não-distribuído), que irá gerar um único autômato para rodar em uma única máquina. A segunda utiliza o comando *machine* e roda de forma distribuída em duas máquinas distintas. Todos os novos comandos encontram-se em negrito.

Primeira Versão: programa RS, sem uso de *machine*.

```
rs_prog mouse:
[ input          : [click, tick],
  output         : [single, double],

        module timer:
        [ input     : [click, tick],
        output      : [start, relax],
        p_signal: [a,b],
        var         : [delta],
        initially   : [up(a)],
                      click#[a] ⇒ [delta:=3, emit(start), up(b)],
                      tick#[a] ⇒ [up(a)],
                      tick#[b] ⇒ case [delta>0 → [delta:=delta-1, up(b)],
                                         else → [emit(relax), up(a)] ]
        ],
        module emitter:
        [ input     : [click, start, relax],
        output      : [single, double],
        var         : [count],
        initially   : [ ],
                      start ⇒ [count:=0],
                      click ⇒ [count:=count+1],
                      relax ⇒ case      [count=0 → [emit(single)],
                                         else → [emit(double)] ]
        ]
].
```

Segunda Versão: o programa RSD, distribuído nas máquinas *sinope* e *pan*.

```
        rsd_prog mouseD:

        external interface:
        [ input     : [click, tick],
         output     : [single, double]
        ].

        machine sinope :
        [ input     : [click, tick],
          output    : [start, relax],
          module timer:
          [ input   : [click, tick],
          output    : [start, relax],
```

```
  p_signal: [a,b],
  var       : [delta],
  initially : [up(a)],
  click#[a] ⇒ [delta:=3, emit(start), up(b)],
  tick#[a] ⇒ [up(a)],
  tick#[b] ⇒ case [delta>0 → [delta:=delta-1, up(b)],
                   else → [emit(relax), up(a)] ]
  ]
].

machine pan:
[ input    : [click, start, relax],
  output : [single, double],
  module emitter:
  [ input : [click, start, relax],
   output : [single, double],
   var : [count],
   initially : [ ],
            start ⇒ [count:=0],
            click ⇒ [count:=count+1],
   relax ⇒ case       [count=0 → [emit(single)],
                      else → [emit(double)] ]
 ]
].
```

3.3.3 O pré-processador RS

Como anteriormente mencionado, a sintaxe de *machine* é a mesma de *rs_prog*. A grande vantagem disso é que o texto compreendido entre duas diretivas *machine* pode ser visto como um programa RS completo. A idéia básica é separar todas as UDs (cada UD é identificada por um comando *machine*) com o uso de um pré-processador, compilá-las e depois enviá-las para a máquina correta – especificada no parâmetro do comando.

O pré-processador tem um trabalho simples. Os passos necessários para transformar cada UD em um programa RS completo estão resumidos abaixo:

1. Separar cada UD em um novo arquivo.
2. Guardar o nome da máquina especificada (parâmetro de *machine*).
3. Substituir *machine* (e seu parâmetro) por *rs_prog Nome*#n, onde *Nome* é o nome do programa distribuído e #n é um contador. O contador é inicializado com o valor 1 e incrementado a cada ocorrência de *machine*.
4. Compilar a UD (já na forma de um programa RS padrão) e enviá-la para a máquina especificada.

Além da facilidade de implementação, uma das grandes vantagens do uso desse método é a possibilidade de se ter várias UDs em uma mesma máquina, pois cada UD terá seu próprio nome, que é único.

3.3.4 Saída do RSD

O Compilador RS gera os autômatos e as regras de todos os programas RS em arquivos para posterior execução no ambiente RS distribuído. Os arquivos possuem o mesmo nome do programa RS (*nome#n*) e são distinguidos por suas extensões: *.rul* identifica um arquivo de regras e *.aut*, um arquivo de descrição de autômato.

Voltando ao exemplo do controlador de *mouse* distribuído, veremos que os seguintes arquivos serão gerados (tabela 3.1 e 3.2):

TABELA 3.1 – Autômato e regras gerados para a *machine* sinope do *mouse* distribuído

mouseD1.aut	**MouseD1.rul**
AUTOMATON "mouseD1" : init - [1, *, go_to(1)] 1 click [2, *, go_to(2)] 1 tick [3, *, go_to(1)] 2 tick [[4 - 1, *, go_to(2)], [4 – 2, *, go_to(1)]]	Rules for "mouseD1" : Module timer: 1. [ ] ===> [ ] 2. [ ] ===> [delta := 3, emit(start)] 3. [ ] ===> [ ] 4. Case: 4-1. [ ] {delta > 0} ---> [delta := delta - 1] 4-2. [ ] {else} ---> [emit(relax)]

TABELA 3.2 – Autômato e regras gerados para a *machine* pan do *mouse* distribuído

mouseD2.aut	**mouseD2.rul**
AUTOMATON "mouseD2" : init - [1, *, go_to(1)] 1 start [2, *, go_to(1)] 1 click [3, *, go_to(1)] 1 relax [[4 - 1, *, go_to(1)], [4 – 2, *, go_to(1)]]	Rules for "mouseD2" : Module emitter: 1. [ ] ===> [ ] 2. [ ] ===> [count := 0] 3. [ ] ===> [count := count + 1] 4. Case: 4-1. [ ] {count = 0} ---> [emit(single)] 4-2. [ ] {else} ---> [emit(double)]

Um outro arquivo é necessário: o Arquivo de Informações Distribuídas (AID), que indica em qual máquina cada autômato deverá rodar, quais os sinais esperados do ambiente externo e quais os sinais que devem ser repassados a outros autômatos. Essas informações não estão representadas nos arquivos *.aut* e *.rul*. O formato do arquivo esta na Figura 3.10:

IOFILE nome do sistema distribuído
External Input: [lista de sinais externos de entrada de todo o sistema]
External Output: [lista de sinais externos de saída de todo o sistema]

AUTOMATON nome do autômato$_1$
External Input: [lista de sinais externos de entrada do autômato 1]
External Output: [lista de sinais externos de saída do autômato 1]
Input From Env.: [lista de sinais que o autômato 1 recebe do ambiente externo]
....
AUTOMATON nome do autômato$_n$
External Input: [lista de sinais externos de entrada do autômato n]
External Output: [lista de sinais externos de saída do autômato n]
Input From Env.: [lista de sinais que o autômato n recebe do ambiente externo]

FIGURA 3.10 – Formato do arquivo de informação distribuída.

Voltando ao exemplo de controlador de *mouse* distribuído, o AID seria como mostrado na tabela 3.3:

TABELA 3.3 – Arquivo de informações distribuídas do *mouse* distribuído

mouseD.iod
IOFILE mouseD External input: [click, tick] External output: [single, double] AUTOMATON mouseD1 (machine *sinope*) External input : [click, tick] External output: [start, relax] Input from env.: [click, tick] AUTOMATON mouseD2 (machine *pan*) External input : [click, start, relax] External output: [single, double] Input from env.: [click]

3.3.5 Distribuição com o uso de nomes reais de máquinas

A forma de distribuir um programa RS em diversos ambientes foi fazer com que o parâmetro do comando *machine*, que identifica a máquina onde a respectiva UD irá rodar, represente nomes de máquinas reais. Assim, o programador, ao fazer o programa RS distribuído, já define em que máquinas serão executadas cada um dos autômatos. A idéia é obter, através dos nomes das máquinas, o endereço IP das mesmas, utilizando o serviço de DNS da rede.

No exemplo do *mouse* distribuído (seção 3.3.2), estabelece-se as máquinas *sinope* e *pan* para executar os autômatos *mouseD1* e *mouseD2*, respectivamente. Nesse caso, no momento em que o controlador de *mouse* for colocado em execução, mais precisamente na hora de inicializar as diversas UDs, cada UD verificará o endereço dos outros autômatos do sistema para o estabelecimento de conexões entre os mesmos.

Vale observar que todas as informações necessárias para a distribuição dos autômatos estão no arquivo AID. Se for alterado, nesse arquivo, os nomes das máquinas em que os autômatos devem rodar, uma nova distribuição para o mesmo programa (isto é, para o mesmo conjunto de autômatos) será obtida.

A possibilidade do uso de um mesmo programa distribuído em diversos ambientes, sem a necessidade de modificar o código-fonte ou mesmo a obrigação de recompilá-lo, torna-se uma ferramenta poderosa para a Linguagem RS, pois pode-se portar qualquer programa a qualquer ambiente (desde que haja o ambiente de execução requerido pela própria linguagem).

3.3.6 O ambiente de execução do RSD

Por enquanto, um programa RSD foi definido como sendo *n* UDs, sendo que cada UD constitui-se de um autômato RS rodando em uma determinada máquina. Um problema que surge nessa organização é a necessidade de algum meio que possibilite às diversas UDs receberem comandos externos, como indicação de inicialização e de finalização, bem como informarem condições internas, como exceções, erros e envio de sinais para o ambiente externo, visto que as UDs poderão encontrar-se em diversas máquinas e não conseguirão, desta forma, acessar o terminal do usuário diretamente para gerar tais informações.

A solução natural do problema envolve a utilização de um sistema tipo mestre-escravo. O mestre envia comandos e sinais para os escravos que, por sua vez, processam os sinais, executam os comandos e enviam o resultado de volta para o mestre que, finalmente, os mostra para o usuário. Por outro lado, a necessidade de simulação do ambiente de execução, no qual o usuário entra com os sinais via teclado, faz com que seja necessário a criação de um outro processo no sistema que seja encarregado exclusivamente da entrada de dados. Esse processo, doravante chamado de RS_IO, tem a função de liberar o mestre do processo bloqueante de entrada de dados. Sua existência está condicionada ao ambiente de simulação, durante a fase de depuração do programa distribuído. Este processo será descartado pelo usuário na hora da implantação definitiva de seu programa reativo, na aplicação que ele se destinar.

Embora bastante simples, a entrada de dados torna-se um encargo demasiado oneroso para o mestre. Por ser um processo bloqueante, o mestre não tem como atender ao teclado e aos escravos (ARSD) simultaneamente. Além disso, a entrada dos sinais pelo teclado ocorre somente no ambiente de simulação, não sendo necessária na implantação final do sistema. Na solução apresentada, na implantação final do sistema, o programador só precisa descartar o RS_IO e direcionar para o mestre a entrada dos sinais reais proveniente do ambiente externo. Na aplicação final, os sinais da interface externa já não representam uma tarefa bloqueante, pois são recebidos pelo mesmo comando que capta os sinais dos ARSDs (os sinais são do mesmo tipo).

Expostas as razões das escolhas, a estrutura do ambiente de execução, com as funções específicas de cada processo, ficou sendo a seguinte:

1. *RS_Main*: encarregado de inicializar e finalizar os ARSDs, repassar cada sinal recebido de RS_IO para os ARSDs, imprimir mensagens de erro e os resultados recebidos de ARSDs, bem como tratar exceções e erros internos. O RS_Main ainda é encarregado de enviar comandos diversos aos ARSDs e ao RS_IO.

2. *RS_IO*: encarregado de receber os sinais do usuário, via teclado, e repassá-los ao RS_Main. Embora a verificação da estrutura e do nome do sinal sejam feitos no próprio RS_IO, o processo não realiza nenhuma operação com o sinal. Quando o sinal é válido, ele é enviado em um formato pré-definido ao RS_Main.

3. *ARSDs*: são as diversas UDs. Realizam os comandos que o RS_Main ordena, tratam os sinais recebidos e enviam seus resultados a quem possa interessar (ou a outros ARSDs ou ao RS_Main).

Voltando ao exemplo do controlador de *mouse* distribuído, apresentado na Seção 3.3.2, e imaginando que o programador esteja simulando seu programa a partir da máquina de nome *Tritão*, a estrutura dos processos será como a mostrada na figura 3.11, onde cada retângulo em branco identifica uma máquina e cada retângulo em cinza identifica um processo.

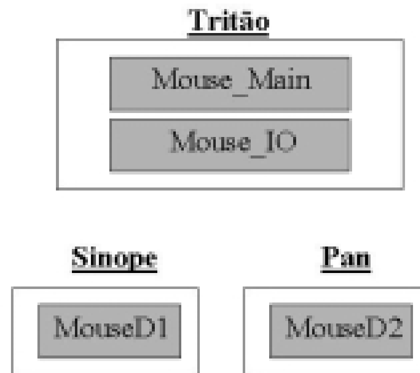

FIGURA 3.11 – Estrutura do ambiente RSD rodando *mouse* distribuído

3.3.7 Comunicação entre autômatos

Para que os autômatos funcionem de forma distribuída, são necessários mecanismos de troca de mensagens, para que o sinal emitido por um autômato seja recebido por aqueles que consideram o sinal relevante.

Uma mensagem é *relevante* para um autômato se o sinal que ela conduz faz parte dos sinais de entrada desse autômato. Assim, se um autômato emite a mensagem X, ela é relevante para todos os autômatos que tenham X em seu conjunto de sinais de entrada. Por exemplo, o módulo RS descrito na figura 3.12 considera as mensagens X e Z relevantes.

```
module exemplo:
[ input: [X, Z],     ⇐ os sinais “X” e “Z” são relevantes para este programa
  output: [A],
 ....................
].
```

FIGURA 3.12 – Relevância de sinais para cada módulo

Mudanças na semântica foram necessárias para que a comunicação entre os autômatos procedesse de forma correta, pois nem todos os sinais *output* declarados no bloco *machine* devem ser realmente emitidos para o ambiente externo. O comando *emit*, que antes significava a emissão de um sinal para o ambiente externo, agora possui também a função de estimular outros autômatos enviando-lhes mensagens que lhes sejam relevantes, ou seja, que estejam em seu conjunto de sinais de entrada. Quando um sinal emitido não é relevante para nenhum dos autômatos, então, ele é enviado para o ambiente externo, pelo processo RS_IO.

3.3.8 Transmissão de sinais para outros autômatos

Sabendo-se o conjunto de sinais que serão consumidos internamente e o conjunto de sinais que serão transmitidos a outros autômatos, pode-se construir uma tabela com essas informações no código gerado.

Os endereços IP dos autômatos, entretanto, só serão conhecidos em tempo de execução. Mas como cada ARSD sabe os nomes reais das máquinas em que os outros ARSDs estão executando, cada um pode descobrir os IPs das demais máquinas. A troca de mensagens entre os diversos processos RS obedece a uma padronização, a qual é vista na próxima seção.

3.3.9 O protocolo RS

O protocolo RS padroniza as informações trocadas a nível RS_Main / ARSD, ARSD / ARSD e RS_IO / RS_Main. Nas seções seguintes, faz-se um estudo sobre a ordenação de mensagens e sua importância e definimos as mensagens que serão trocadas entre os diversos processos do sistema.

3.3.9.1 A importância da ordenação total de mensagens

No caso geral de um sistema formado por vários módulos distribuídos em diversas máquinas, que enviam sinais relevantes uns para os outros, se não houver uma ordenação para as mensagens, o sistema pode retornar não determinismo. Para evitar tal fato, uma solução seria fazer com que as mensagens passassem por algum tipo de ordenação. Pode-se usar, por exemplo, o algoritmo de *Cristian* [TAN 95] [STA 95] ou então implementar-se o relógio lógico de *Lamport* [PET 85] [SIL 88], para garantir que as mensagens sejam tratadas na ordem em que elas são enviadas.

Algoritmos de sincronização de relógios físicos, como o de *Cristian*, não são adequados par dois motivos. Primeiro, porque a sincronização física de relógios exige uma grande troca de mensagens, gerando um tráfego de rede pesado. Segundo, porque o tempo real não é realmente relevante. O que importa é saber se uma mensagem foi enviada antes do que outra.

Algoritmos que utilizam relógios lógicos, como o de *Lamport*, parecem encaixar-se perfeitamente no caso da linguagem RSD, pois praticamente não causam *overhead* e solucionam o problema do indeterminismo. Entretanto, nem todas as mensagens necessitam ser ordenadas e, novamente, seria um desperdício de banda marcar todas as mensagens com um *time-stamp*. Além disso, muitas bibliotecas de distribuição, como o MDX, garantem que a ordem local das mensagens seja respeitada, isto é, se um processo A envia duas mensagens para o processo B, a ordem com que as duas mensagens serão recebidas será idêntica à ordem que foram enviadas.

Um *time-stamp* de 32 bits garante que os processos funcionem com uma ordenação total por um período de tempo relativamente grande. Por exemplo, em um sistema que tenha uma troca média de 100 mensagens por segundo (um número razoavelmente elevado), o sistema todo funcionaria corretamente por 1,36 ano. Depois disso, seria necessário a redefinição do relógio lógico coordenado pelo RS_Main.

O relógio lógico usado em RSD foi definido, baseado nas observações acima, da seguinte forma:

1. Apenas as mensagens que realmente necessitam ser ordenadas possuem um *time-stamp*, economizando banda de rede e o próprio contador do relógio. Essas mensagens são RS_ACK, RS_NACK, RS_PRINT, RS_EMIT e RS_PING.
2. Quando o relógio lógico chega a 4.294.000.000, o RS_Main envia um comando para todos os ARSDs, exigindo que estes redefinam seus relógios para zero.

Desta forma, garante-se o uso do relógio de Lamport por um tempo infinitamente grande.

3.3.9.2 As mensagens do protocolo RS

O protocolo RS define as mensagens e a forma com que elas são tratadas. As definições são:

1. RS_ACK e RS_NACK: utilizadas para identificar que o sinal foi compreendido (RS_ACK) ou que, por alguma razão, ele não foi lido ou aceito (RS_NACK).
2. RS_CLOCK: mensagem que tem o objetivo de redefinir os relógios lógicos. Enviada somente pelo RS_Main, força que os ARSDs redefinam seus relógios lógicos para zero, enviem um RS_ACK e fiquem esperando sincronamente uma mensagem RS-REINIT.
3. RS_PRINT: enviada pelos ARSDs para o RS_Main. Tem como função imprimir uma mensagem qualquer no console de saída.
4. RS_EXCP: mensagem enviada pelos ARSDs, indicando uma condição interna de erro. O ARSD que a emite fica esperando sincronamente um RS_REXCP, que decidirá se o ARSD deve continuar executando ou deve ser finalizado.
5. RS_EXIT: possui dois sentidos. Se enviado pelo RS_Main, significa uma ordem para que o processo destinatário finalize – não pode ser negado. Se enviado por qualquer ARSD ou pelo RS_IO, é um pedido de finalização e, por conseqüência, pode ser negado pelo RS_Main.
6. RS_REXCP: mensagem de resposta a RS_EXCP. Carrega somente um parâmetro que pode assumir dois valores. Se for igual a zero, o processo que enviou RS_EXCP pode (e deve) finalizar; se for igual a 1, deve continuar com o processamento normalmente.
7. RS_EMIT: principal mensagem da linguagem RSD. Pode indicar duas ações. Se o destino da mensagem for o RS_Main, indica a emissão de um sinal para o ambiente externo. Se o destino for um ARSD, a mensagem indica o simples repasse de uma mensagem para outro ARSD.

8. RS_INIT: enviada pelo RS_Main a todos ARSDs e ao RS_IO. Quando o processo recebe a mensagem, ele deve obrigatoriamente enviar um RS_ACK, avisando ao RS_Main que a mensagem foi entendida e que ele já está pronto para realizar suas tarefas.
9. RS_PING: possui a mesma função do comando do Unix. Pode ser enviado por qualquer processo e um RS_ACK será a resposta positiva, indicando que o ARSD inquirido está “vivo”. Pode ser usado também para atualização do relógio lógico.
10. RS_STOP: enviado pelo RS_Main aos ARSD. É um pedido para que o processo pare todo o processamento, envie um RS_ACK identificando que entendeu a mensagem e fique bloqueado à espera de um RS-REINIT.
11. RS_REINIT: força o autômato a reiniciar suas atividades do ponto em que havia parado. O mestre fica esperando uma resposta RS_ACK síncrona.
12. RS_INPUT: mensagem usada somente para passar o sinal digitado pelo usuário em RS_IO para o RS_Main.

4 O Compilador RS

O compilador RS 5.0, foi desenvolvido em C-ANSI, e portanto, é multiplataforma. O compilador recebe como entrada um arquivo com um programa fonte RS, que pode ser um programa RS distribuído, um programa RS centralizado ou um módulo RS. O programa passa pelas análises sintática e semânticas e não havendo erros, passa para a etapa de geração de código.

Em relação ao código gerado, é mantida a saída do compilador RS original (RS 4.0), isto é, a geração dos arquivos *autômato* e *regras*. Além deste, é possível gerar um programa C, padrão ANSI, e ou um arquivo com código no formato portável OC [COU 98], que está se tornando padrão como um formato comum para representação dos autômatos de estados finitos associados às linguagens reativas síncronas.

O formato OC é usado pelas linguagens Lustre e Esterel, entre outras e permite tradução eficiente em várias linguagens de programação, como por exemplo, C, ADA, EMC, etc., usando pós-processadores apropriados. O código OC gerado pelo compilador RS 5.0 tem as mesmas características de arquivos gerados em Esterel e Lustre.

Na geração de código C para sistemas distribuídos, definiu-se que dois processos sempre serão gerados [GIO 98]: o RS_Main e o RS_IO. Conforme foi visto no capítulo interior, o primeiro tem a função de receber sinais proveniente do RS_IO e repassá-los aos autômatos, bem como gerenciar o sistema, cuidando da sua finalização, casos de pane, inicialização dos ARSDs, etc. O segundo processo, RS_IO, tem a tarefa de receber os sinais digitados pelo usuário e repassá-los ao RS_Main. RS_IO só existe em tempo de depuração, podendo ser descartado quando da instalação real do sistema, momento em que os sinais do ambiente serão enviados diretamente ao RS_Main. Isso indica que este sistema segue os moldes de sistemas mestre-escravo. Além destes processos, será gerado um programa C, com primitivas de comunicação MDX, para cada comando *machine* especificado no código fonte RS 5.0.

4.1 A Construção do Compilador RS

Com exceção da parte de geração de código, o compilador foi construído utilizando técnicas de compilação tradicionais. O Analisador Léxico foi desenvolvido manualmente (isto é, sem utilizar a ferramenta Lex [LESK 74]) e o Analisador Sintático foi programado utilizando a técnica de Análise Preditiva Tabular [PRI 2000]. A maior parte do trabalho esteve concentrada na obtenção da tabela de análise LL(1), a partir da gramática de RS. Aqui será apresentada apenas a parte de geração de código, que não é convencional.

4.2 Gerador de Código

O gerador de código é acionado após as etapas da análise, quando o programa está sintatica e semanticamente correto.

No caso da Linguagem RS 5.0, o código gerado é o autômato RS. Este código intermediário dá origem aos dois formatos de arquivos, ou seja, o arquivo C, para a simulação do autômato, e o arquivo no formato portável OC, que torna o código RS compatível com as linguagens LUSTRE e Esterel. Nas próximas seções, são mostrados o processo de geração dos autômatos e os tipos de formatos de arquivos gerados pelo compilador RS 5.0.

4.3 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 seqüência de transições que disparam quando a rede, no estado de espera *n*, é estimulada por um sinal *s*, determina a seqüência de ações que o autômato deve executar quando, no estado *n*, é estimulado por *s*. Esta seqüência é formada por subseqüências 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 seqüência 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 seqüências, 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 (cada reação é uma seqüência de ações acompanhada de uma mudança de estado) 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.

4.4 O Código C para a simulação de Autômatos RS Distribuídos

Nesta seção, serão explicadas as principais características do código C gerado pelo Compilador RS 5.0. Detalhes de implementação, como nomes de funções e chamadas a sub-rotinas, serão deixados de lado para nos concentrarmos, especificamente, nos seguintes detalhes:

1. Declaração e tipificação de variáveis e sinais;
2. Tratamento de asteriscos;
3. Representação do autômato em C;
4. Tratamento de exceções internas;
5. A interface com o usuário;
6. A execução do sistema;
7. Implementação do protocolo RS.

A geração do código C passa por duas fases distintas: a geração de regras e a geração do autômato. Na primeira fase, são identificadas todas as variáveis. A tradução de comandos e atribuições RSD para a linguagem C também é feita nesse momento. Em uma segunda fase, a ordem de execução das regras é definida. Partindo-se da definição do autômato RS, constrói-se a rotina *autômato* do código C, que basicamente conterá chamadas a procedimentos e funções anteriormente traduzidos.

4.4.1 Declaração e tipificação de variáveis e sinais

Como não existe declaração de tipo em RS, o compilador define todas as variáveis do sistema como do tipo *int.* Assim, portanto, toda e qualquer variável declarada em um programa fonte RSD, vai ser interpretada no programa RSD gerado como uma variável inteira.

4.4.2 Tratamento de asteriscos

Os asteriscos, na representação de um autômato RS, têm o objetivo de separar ações que podem ser executadas em paralelo. Para garantir o correto funcionamento dessas ações, deve-se garantir que todas elas terão acesso aos mesmos valores de sinais. Isso significa que as mudanças que uma ação realizar em um sinal X não serão visíveis em uma ação paralela. Entretanto, no fim do passo de reação, marcado pelo próximo asterisco, o valor deve ser corretamente atualizado. Em relação ao uso de variáveis, o compilador RS nunca deixará que duas ações, dentro de um mesmo passo de reação, modifiquem a mesma variável.

Para realizar o controle sobre os valores dos sinais, usamos o mesmo esquema adotado pelo compilador RS 4.0. Nele, uma cópia extra do sinal é alocado. Valores sempre são lidos do sinal original e escritos na cópia. Quando o passo de reação acaba, a cópia é usada para atualizar o sinal original. Convém ressaltar que o compilador RS 5.0 verifica se uma segunda escrita está sendo feita sobre a cópia da variável. Caso seja uma nova escrita e os valores, da primeira e da nova, sejam diferentes, uma situação de inconsistência ocorreu. Então, o compilador acusa um erro de execução.

4.4.3 Representação do autômato em C

Para exemplificar a codificação de um autômato RSD, vamos nos basear no exemplo do *mouse* distribuído (seção 3.3.2). O trechos abaixo, foram retirados dos arquivos *mouseD1.aut* e *mouseD1.rul*. Eles correspondem ao autômato *mouseD1* e representa o que deve ser feito quando o autômato está no estado 2 e ocorre o estímulo *tick*. A variável *delta* representa a quantidade de *tick*s provenientes do ambiente externo. Quando *delta* for maior que 0, o seu valor será decrementado em uma unidade e o autômato continuará no estado 2. Quando delta for igual a 0, o sinal *relax* será emitido e o autômato passará para o estado 1.

(Arquivo *mouseD1.aut*)

```
2   tick  [[4 – 1, *, go_to(2)], [4 - 2, *, go_to(1)]]
```

(Arquivo *mouseD1.rul*)

```
4.  Case:
  4-1. [ ] {delta > 0} ---> [delta := delta - 1]
  4-2. [ ] {else} ---> [emit(relax)]
```

A seguir é apresentado o código C correspondente aos trechos anteriores. Note que a variável *user* tem o sinal que vem do ambiente externo. A função *RS_EMIT* é responsável por emitir o sinal *relax* para quem está o esperando, no caso, o autômato *mouseD2*.

```
if(est_atual == 2)
if(!strcmp(user, "tick")) {
      if(delta > 0) {
         delta = delta - 1;
         est_atual = 2;
      }
      else {
         RS_EMIT("relax");
         est_atual = 1;
      }
}
```

Embora a representação dos autômatos RSD seja bastante simples, foi necessário algum esforço para a construção de um compilador geral que permitisse o tratamento dos diversos níveis de aninhamento que podem ocorrer nas ações. Por exemplo, pode existir um comando de teste em que uma das opções, o qual se desdobra em outro teste, que se desdobra em outro teste, e assim por diante, sem previsão do número máximo níveis.

4.4.4 Tratamento de exceções internas

Considera-se exceção interna qualquer falha na comunicação ou na própria execução, como, por exemplo, a falta de uma ação em um *case*. Tal função emite uma mensagem *RS_EXCP* ao RS_Main e fica esperando uma resposta, que poderá ser uma ordem de finalização ou um pedido para que continue o processamento normalmente.

A espera pela resposta é síncrona, de forma que o ARSD não executará comando algum até a chegada da resposta. Desta forma, RS_Main é obrigado a respondê-la em tempo hábil, para que não ocorra prejuízo no processamento. Na versão atual, a resposta sempre será um pedido para continuar o processamento. Entretanto, pode-se, a critério do programador, inventar outras exceções e criar decisões mais complexas sem a necessidade de reprogramação dos escravos – apenas do mestre.

4.4.5 A interface com o usuário

A interface do o usuário com o Compilador RS 5.0 é feita através de parâmetros passados diretamente na linha de comando, parâmetros esses que indicam o nome do arquivo (programa) a ser compilado e a máquina onde os processos RS_Main e RS_IO executarão. Por exemplo, o comando

```
%> crs mouseD tritao
```

indica que deve ser compilado o programa que está no arquivo *mouseD* e que os processos RS_Main e RS_IO devem ser carregados na máquina *Tritão*. Como o programa tem duas declarações *machine*, o compilador irá gerar dois autômatos. Após a compilação, terão sido gerados os arquivos *Mouse_IO.c*, *Mouse_Main.c, MouseD1.c,* e *MouseD2.c*. O método da execução dos mesmos está descrito abaixo.

O segundo parâmetro (o nome da máquina para os processos RS_Main e RS_IO), é necessário, pois em nenhum momento isto foi citado no código RS. Caso esse parâmetro não seja especificado pelo usuário, o compilador solicitará o nome de uma máquina para desempenhar o papel de máquina principal do sistema distribuído (a máquina que executará os processos centrais de um ARSD). Para o caso de uma compilação de um autômato não distribuído, o segundo parâmetro é desnecessário, pois o autômato não tem que se comunicar com nenhum outro processo e pode ser executado em qualquer compilador C, padrão ANSI.

4.4.6 Execução do sistema

Para a execução de um ARSD, é necessário um certo trabalho. É necessário estar logado em cada máquina do ambiente especificado no código RS distribuído e também na máquina especificada no parâmetro passado ao compilador, para a compilação e execução de cada autômato. No caso do exemplo que estamos considerando, o usuário deve estar ativo nas máquinas *tritão* (*Mouse_Main* e *Mouse_IO*), *sinope* (*mouseD1*) e *pan* (*mouseD2*). Estes arquivos devem ser compilados e executados em cada uma das máquinas, isso porque o código é dependente de cada sistema operacional.

Todos os ARSDs e o *Mouse_IO* passam por uma rotina de inicialização, supervisionados pelo *Mouse_Main*. Se algum processo for considerado morto ou não-confiável, uma mensagem de aviso será enviada ao usuário e o sistema será finalizado, pois a condição mínima para a execução de um programa RSD é que todos os processos estejam executando a função que lhes foi atribuída.

O *Mouse_Main* enviará um *RS_INIT* para cada autômato (*mouseD1* e *mouseD2*) e ficará bloqueado até que cada um responda com um *RS_ACK*. Se ambos responderem, envia uma solicitação de inicialização para o *Mouse_IO*. Recebida um *RS_ACK* do *Mouse_IO*, o mesmo estará disponível ao usuário para a simulação do autômato. Após a execução de todos os arquivos, o usuário vai no terminal onde está sendo executado o *Mouse_IO*. Nele é que o usuário entrará com os sinais de entrada e receberá as respostas do sistema.

4.4.7 Implementação do protocolo RS

A implementação do protocolo RS está praticamente toda centralizada. Embora as primeiras mensagens *RS_INIT* sejam tratadas no início do processo e, portanto, fora do controle central, existe, inclusive para elas, um tratamento na parte centralizada. A razão principal da implementação de um controle central foi a facilidade de manutenção. Sendo identificadas as mensagens em um único local, pode-se localizar o procedimento que a trata e, se uma nova mensagem for inserida, será necessário tão somente a inclusão de um novo teste e de um novo procedimento para tratá-la.

Sendo possível aceitar qualquer mensagem, foi implementada uma rotina de exceção que recebe inclusive mensagens não previstas. Nessa rotina, mensagens não esperadas são avisadas ao RS_Main que, por sua vez, avisa ao usuário que decide se o processo deve continuar. Por exemplo, se uma mensagem *RS_INIT* for recebida no meio do processamento, será enviado um aviso ao usuário e o processo que o recebeu perguntará à RS_Main se deve continuar com o processamento.

4.5 O Formato OC

As principais linguagens baseadas na hipótese do sincronismo são Esterel [BER 92], Lustre [HAL 91], Statecharts, SML, Signal, Saga e Argos. Elas são compiladas em autômatos finitos, que podem ser implementados em várias linguagens de programação clássicas, como por exemplo, C, ADA, EMC, etc. Para isso, é natural que seja definida uma descrição abstrata de tais autômatos, utilizando um formato comum a todas as linguagens. Buscando essa padronização, surgiu o formato portável OC (*Object Code*), que permite tradução eficiente em várias linguagens de programação, usando pós-processadores apropriados. OC é um formato comum de saída dos compiladores Esterel, Lustre e Argos. Recentemente este formato também foi adaptado por Signal e Saga.

Um arquivo OC descreve uma lista de módulos, que são nodos LUSTRE ou módulos de Esterel. O nome de um arquivo deste tipo deve ser posfixado por ".oc". Um módulo consiste em um cabeçalho de módulo, uma série de tabelas que descrevem os objetos que o autômato pode referir, um autômato e um terminador. O cabeçalho indica em qual versão do formato portável, foi escrito o módulo.

Por exemplo, um código OC escrito na sua versão 5:

```
oc5:
module: WATCH
        tabelas
        autômato
end module:
```

O formato portável é projetado para ser usado por pós-processadores que traduzirão os autômatos para uma linguagem de programação seqüencial. Esta seção descreve o formato dos arquivos OC gerados pelo Compilador RS 5.0, isto é, o formato requerido por tais processadores. O formato também pode acomodar extensões para satisfazer as exigências de outros processadores. Por exemplo, um depurador ou uma verificação de sistema precisa de informações extensas sobre os nomes de variáveis e sinais, como também os valores levados por estas entidades durante a execução de um módulo. Este tipo de informação é altamente dependente da linguagem.

4.5.1 As tabelas

Os objetos referidos pelo autômato estão dentro de várias classes, por exemplo, tipos, variáveis, cada um associado com uma tabela. Dentro de cada classe, pode haver alguns objetos pré-definidos, por exemplo, o tipo inteiro e a função *mod*. Cada tabela descreve só os objetos definidos pelo usuário de sua classe. Todavia, os objetos pré-definidos podem ser referidos em qualquer lugar nas tabelas ou no autômato.

Cada tabela começa com uma linha de cabeçalho, identificando a classe de objeto e mostrando o número de entradas, e termina com "end:". Por exemplo:

```
variables: 13
        ...
end:
```

Se a tabela não está vazia, o cabeçalho é seguido por uma série de linhas de entrada, uma por objeto. Cada linha de entrada pode conter um número arbitrário de sinais de retorno, tabulações e espaços em branco.

Há 13 tabelas que aparecem em uma ordem fixa. Um conjunto típico de tabelas é: *instances*, *types*, *constants*, *functions*, *procedures*, *signals*, *implications*, *exclusions*, *variables*, *tasks*, *execs*, *actions* e *halts*.

Cada linha de entrada consiste de um índice seguido por um ":" e uma entrada. Um índice identifica unicamente uma entrada em uma dada tabela, assim pode-se usar livremente o índice para designar a entrada. Pode-se falar, por exemplo, sobre "variable 5", ou "function function-index".

Índices vão de 0 a n, onde n é o número de entradas dado em um cabeçalho (exceto nos casos da tabela de *halt*, onde o índice vai de 1 para n). Dependendo da entrada, o índice refere outros objetos, sendo no caso os atributos do mesmo. Se o objeto referido é pré-definido, então seu índice deve ser precedido por $. Por exemplo, a entrada para um inteiro constante THREE contém:

```
THREE $3
```

As tabelas do formato OC estão descritas abaixo:

- **Tabela de instâncias:** Essa tabela lista todas as instruções executáveis que aparecem em um programa Esterel.
- **Tabela de Tipos:** Para cada tipo *type_name*, é assumido que existe um procedimento *_assign_type-name*, uma função de igualdade *_eq_type_name*, uma função de diferença *_ne_type_name*, e uma função condicional *_cond_type_name*. Se uma dessas funções não é usada no módulo, o campo correspondente é void.
- **Tabela de Constantes:** É a tabela que contém todas as constantes, seus nomes, tipos e valores.
- **Tabela de Funções:** Esta tabela contém os nomes das funções, os tipos dos argumentos e o tipo de retorno. É uma lista indexada, isto é, uma lista de tipos indexados separados por vírgula entre parênteses; *Type equality functions*: para cada tipo *type-name*, há implicitamente declarada uma função chamada *_eq_type-name*. Essa função pega dois argumentos do tipo *type-name* e retorna um resultado booleano, verdadeito se os dois argumentos são iguais, falso de outra forma. *Type difference functions*: para cada tipo *type-name*, há implicitamente declarada uma função chamada *_ne_type-name*. Essa função pega dois argumentos do tipo *type-name* e retorna um resultado booleano, verdadeiro se os dois argumentos são diferentes, falso se não. *Type conditional functions*: para cada tipo *type-name*, há implicitamente declarada uma função chamada *_cond_type_name*. Essa função pega um argumento de tipo booleano, dois argumentos do tipo *type-name*, e

retorna um resultado do tipo *type-name*. Ele computa a expressão condicional do tipo *type-name*.

- **Tabela de Procedimentos:** Tabela que contém os nomes dos procedimentos. Para cada procedimento, tem-se uma lista indexada, a qual dá os tipos dos argumentos passados por referência e uma lista indexada que dá os tipos dos argumentos passados por valor. Os procedimentos *type assignment*: para cada tipo *type-name*, há implicitamente declarado um procedimento chamado *_assign_type-name*. Esse procedimento pega dois argumentos do tipo *type-name*, um por referência (a variável a receber o valor) e um por valor (a expressão avaliada).

- **Tabela de Tarefas:** Essa tabela é vazia no caso de código compilado de programas LUSTRE. Cada linha possui um nome de tarefa, uma lista indexada que dá os tipos de argumentos passados por referência e uma lista indexada que dá os tipos de argumentos passados por valor;

- **Tabela de Execução:** Essa tabela é vazia em caso de códigos compilados de programas LUSTRE. Ela registra todas as chamadas à tarefas externas que são feitas pela instrução ESTEREL *exec*. Há uma entrada nessa tabela para cada ocorrência da instrução *exec* no código fonte ESTEREL. Indica a tarefa chamada, o índice do sinal de retorno associado com o *exec*, a lista de argumentos passados por referência na chamada e a lista de argumentos passados por valor da chamada.

- **Tabela de Sinais:** Sinais são usados para transmitir valores, sincronizar e acessar parâmetros externos assincronamente. Há duas classes de sinais: visíveis e invisíveis. Os sinais visíveis são usados para controlar a interface entre o módulo e o ambiente. Em ESTEREL, eles são o input, output, inputoutput e return signals, e os sensores os quais não desempenham qualquer papel na sincronização, mas são usados para ler assincronamente valores modificados. Em LUSTRE, eles são as variáveis input e output. Os sinais invisíveis são usados para comunicação interna.

- **Tabela de Implicação:** Uma implicação relaciona dois sinais, o escrevo e o mestre. A presença do mestre implica na presença do escravo. Em Esterel, uma implicação é escrita *master=>slave*. Em LUSTRE, o *clock* mestre é um *subclock* do *clock* do escravo.

- **Tabela de Exclusões:** Uma exclusão consiste de uma lista de sinais mutuamente exclusivos, isto é, que não podem estar presente ao mesmo tempo.

- **Tabela de Variáveis:** A sintaxe para uma entrada é: *type-index [value: initial-value]* onde *type-index* é o seu tipo. Se o campo "*value:*" ocorre, a variável é inicializada com "*initial-value*", que deve ser uma expressão constante, isto é, uma expressão envolvendo nenhuma variável. A inicialização de variáveis do módulo *module* ocorre quando o módulo é chamado pela primeira vez e cada vez que o usuário reinicializa o módulo pela invocação do procedimento especial *module_reset*.

- **Sintaxe das expressões:** Ações comumente envolvem expressões; para melhor compreensão das entradas na tabela de ações, as expressões são explicadas aqui. Uma expressão em LUSTRE ou Esterel consiste de átomos (inteiros, floats, doubles e strings), constantes, variáveis, sensores e chamadas de funções. Em LUSTRE, uma expressão pode também ser uma referência para um campo de uma tupla. Uma expressão constante é uma que não contém variável ou sensores. Átomos são precedidos por '#'. Constantes são

representadas por seu índice, precedida por '@'. Variáveis são representados por seu índice. Sensores são representados por seu índice, precedido por '?'. Chamadas de função tem a seguinte forma:

function-index(expression-list)

onde *expression-list* é uma lista de expressões separadas por vírgula.
Campos de referência tem a seguinte forma

expression.field-name

onde *expression* é uma expressão e *field-name* é o nome do campo atual, isto é, um identificador LUSTRE. É claro, uma referência igual faz sentido somente se o tipo da expressão é um tipo tupla com um campo chamado *field-name*.

- **Tabela de Ações:** Essa tabela especifica todas as ações elementares que o autômato pode executar durante as transações. Uma entrada dessa tabela é uma lista não-vazia de ações (sem separador) para serem executadas em seqüência. Ações são de três tipos. Ações de teste sempre são seguidas por dois possíveis caminhos de execução. Ações lineares não afetam o fluxo de controle de uma transição. Há uma ação especial *newstate*, a qual é executada pelo autômato quando ele termina uma transação e alcança um novo estado. Há três ações de teste (*present*, *if* e *dsz*), há 10 ações lineares (*call*, *output*, *reset*, *combine*, *start*, *kill*, *return*, *suspend*, *activate*, *act*) e a ação *newstate* (*goto*).

- **Tabela de Halt:** Essa tabela é vazia em caso de código compilado de programas LUSTRE. Ela registra o uso de informações sobre a instrução *halt* no código intermediário Esterel. Ela é usada somente para depuração e relatório de erros e pode ser inteiramente dispensada se isto for conveniente. Há uma entrada por instrução *halt* no programa original. Após o índice de cada entrada vem o nível da instrução *halt*, sendo que esses níveis crescem (são numerados) a partir. Uma tabela típica de *Halt* é a seguinte:

```
halts: 3
1: %lc: 12 42 0%
2: %lc: 245 2 1%
3: %lc: 12 2 1%
```

4.5.2 O autômato

Um autômato é completamente descrito por transições de estados. Essas transições consistem de uma série de ações. Ações de teste causam uma bifurcação binária na estrutura da transição, seguindo para o caminho do sucesso ou falha do teste. No fim de todo caminho, há a indicação do próximo estado do autômato. Uma estrutura de transição de um autômato pode então ser representada, em caso geral, por um grafo acíclico, ou *dag*.

No formato OC, o autômato é descrito por um conjunto de *dags*. Cada estado do autômato é dado por uma codificação *dag* de todas as possíveis transições desse estado. Um *dag* pode também referir outros *dags* que não representam estados, mas que podem ser referenciados por vários estados, ou que podem ser referenciados pelos mesmo estados várias vezes. Isso permite uma representação mais compacta de um autômato. Isso é somente usado na forma otimizada do formato. Esses *dags* que não representam estados são utilizados só quando o código gerado é otimizado (formato otimizado). Os *dags* compartilhados são

agrupados dentro da tabela *dags*:, enquanto o *dag* associado com estados são agrupados dentro da tabela *states*:. Essas tabelas aparecem nesta ordem.

Codificação da transação do estado

Entende-se por *dag* fechado, os *dags* que referem para próximos estados, caso contrário, o *dag* é dito aberto. Assim, dado um *dag* associado com um estado, substituindo recursivamente todas as referências para um *dag* pelo *dag* referido, o resultado final será um *dag* fechado.

Os nodos *dags* podem ser de três tipos:

- um nodo de ação: é uma ação para executar. Se essa ação é um teste, então o nodo tem dois filhos, senão tem somente um. Um nodo de ação contém o índice da ação na tabela de ações.
- um nodo de referência: refere um *dag* para usar numa execução contínua.
  Há dois tipos de codificação para um nodo de referência:
  – '{' *dag-index* '}' se o *dag* do índice dag-index um *dag* aberto.
  – '[' *dag-index* ']' se o *dag* do índice dag-index um *dag* fechado.
- um nodo estado: dá o índice do próximo estado a ser assumido. É codificado como '<' *state-index* '>'.

Outras regras de codificação são:

- nodos vazios são ignorados.
- nodos com somente um filho são escritos diretamente.
- para teste de ações, o índice de ação é escrito primeiro, seguido pela ramificação verdadeira entre parênteses, então pela ramificação falsa entre parênteses e finalmente pela possível continuação.

A tabela de dags

Essa tabela é opcional e somente aparece quando se tem produção otimizada de OC. A sintaxe para um entrada é semelhante a uma gramática livre do contexto:

```
dag      : action_tree
         | open_dag ';'
```

Como as outras tabelas, a tabela *dag* começa com um *header*:

```
dags     : table-size
calls    : call-number
```

call-number é o número de chamadas de ações feitas por todos os *dags*.

A tabela de Estados

A tabela de estados lista todas os possíveis estados para o autômato. Cada entrada representa é um estado e a sintaxe é a seguinte:

state-index : action_tree

Após a expansão do *dag* correspondente, *action_tree* se torna um *dag* fechado. A tabela de estados começa com:

states : *number-of-states*
startpoint: *state-index*
sink : *state-index*
calls : *number-of-calls*

onde o *number-of-states* é o número de estados do autômato, e *startpoint* é o índice do estado inicial, *sink* é o índice de um estado morto e *number-of-calls* é o número total de micro passos em todas as transições e. O campo *sink* é opcional.

4.8.3 Exemplo de Código OC

Nesta seção, é mostrado o código no Formato OC referente ao programa RS que mostra o funcionamento de um mouse, o qual está descrito na seção 3.2.2.

```
oc5:

module: MOUSE

instances: 1
root: 0
0: MOUSE 0 "" "rato.rs" %lc: 1 1 0% %lc_end: 22 7 0%
end:

signals: 6
0: input: CLICK - pure: bool: 0 %previous: first:% %lc: 2 9 0%
1: input: TICK - pure: bool: 1 %previous: 0% %lc: 2 16 0%
2: output: SINGLE - pure: %previous: 1% %lc: 3 10 0%
3: output: DOUBLE - pure: %previous: 2% %lc: 3 18 0%
4: local: pure: %name: RELAX% %previous: 3% %lc: 4 10 0%
5: local: pure: %name: START% %previous: 4% %lc: 4 17 0%
end:

variables: 3
0: $0 %sigbool: 0% %lc: 2 9 0%
1: $0 %sigbool: 1% %lc: 2 16 0%
2: $1 %count: % %lc: 6 18 0%
end:

actions: 6
0: present:0 %lc: 2 9 0%
1: present:1 %lc: 2 16 0%
2: output:2 %lc: 3 10 0%
3: output:3 %lc: 3 18 0%
4: call:$1 (2) (#4) %lc: 6 18 0%
5: dsz:2 %lc: 6 18 0%
end:

halts: 6
```

```
0: %lc: 22 7 0%
1: %lc: 6 12 0%
2: %lc: 5 8 0%
3: %lc: 11 12 0%
4: %lc: 14 15 0%
5: %lc: 18 15 0%
end:

states: 5
startpoint: 1
sink: 0
calls: 25
%computed: awaited:%

0: %haltset: 0%
<0>

1: %haltset:%
<2>

2: %haltset: 2 3%
0 (4 <3>%emitted: 5%) ()
<2>
%awaited: 0%

3: %haltset: 1 4%
0 (1 (5 (3 <2>%emitted: 4%) ()
<4>) ()
<4>) ()
1 (5 (2 <2>%emitted: 4%) ()
<3>) ()
<3>
%awaited: 0 1%

4: %haltset: 1 5%
1 (5 (3 <2>%emitted: 4%) ()
<4>) ()
<4>
%awaited: 1%

end:

endmodule:
```

5 O Ambiente de Execução RS Distribuído

Neste capítulo, é discutida a implementação do ambiente de execução de autômatos RS distribuídos e justificada a decisão de realizar a implementação em C usando o novo núcleo de comunicação MDX (*Multimídia Distribuída em UNIX*), o qual provê troca de mensagem ao invés de uma memória virtual distribuída.

O objetivo do ambiente de execução de ARSD é complementar o compilador. A idéia é, a partir dos autômatos gerados por este, gerar o código C com chamadas para primitivas MDX. Sua portabilidade fica amarrada ao MDX, portanto. Entretanto, permite ao programador a distribuição do controle ou de qualquer núcleo reativo.

5.1 O Ambiente de Programação Paralela MDX

O sistema MDX permite a criação dinâmica de *threads* e oferece a abstração de uma memória compartilhada. A idéia principal é permitir que o programador escreva um programa *multithread* no qual as *threads* são criadas na rede de processadores, e o compartilhamento dos dados dá-se através de uma memória virtual distribuída.

O ambiente MDX é um sistema que executa sobre uma rede heterogênea de estações de trabalho, abstraindo-a como uma máquina virtual paralela, onde cada nodo da rede representa um computador, composto por um ou mais processadores e uma memória local, interconectados por uma rede física, sobre os sistemas operacionais Windows, OS/2, Linux ou Solaris, entre outros.

O sistema é baseado no modelo cliente/servidor. Nesse modelo, os serviços são oferecidos por servidores especializados. Um programa cliente solicita a realização de um serviço para um servidor enviando os parâmetros adequados. O cliente é suspenso e a execução do serviço começa. Ao término da execução do serviço os resultados são enviados para o programa cliente que é acordado [COP 2000].

Uma qualidade importante deste esquema é que ele funciona independentemente da localização do cliente e do servidor. Se estes são localizados na mesma máquina a comunicação ocorrerá localmente, senão, a comunicação se realizará através da rede.

5.1.1 Modelo de programação

A programação no ambiente MDX é realizada utilizando o modelo de memória distribuída compartilhada [LI 88] com a programação SPMD (*Simple Program Multiple Data*), com o mesmo programa replicado em todos os nodos da rede e execução de um determinado trecho de código (*thread*) em cada um deles. A sincronização é feita com o uso de semáforos e barreiras.

O ambiente MDX tem como principais características a criação dinâmica de *threads* em qualquer processador, a sincronização com semáforos e barreiras, a comunicação através de uma memória virtual distribuída e a implementação baseada no modelo cliente-servidor com a utilização de RPC [COS 93].

A programação no ambiente dá-se através de um conjunto de primitivas de uma biblioteca portável desenvolvida em C e C++, que deve ser compilada e ligada com o programa através dos compiladores de cada sistema operacional nativo.

5.1.2 Implementação

O Sistema MDX é executado sobre um núcleo de comunicação. Esse sistema é formado por um Servidor de Nomes (SN), um Gerenciador de Memória Virtual Distribuída (GM), um Gerenciador de Compilação (GC), um Gerenciador de Execução (GE), um Gerenciador de Sincronização (GS) e um Gerenciador de Programas (GP), conforme a figura 5.1.

FIGURA 5.1 – Visão do Sistema MDX

A arquitetura básica do sistema MDX sobre uma rede de estações de trabalho é apresentada na figura 5.2.

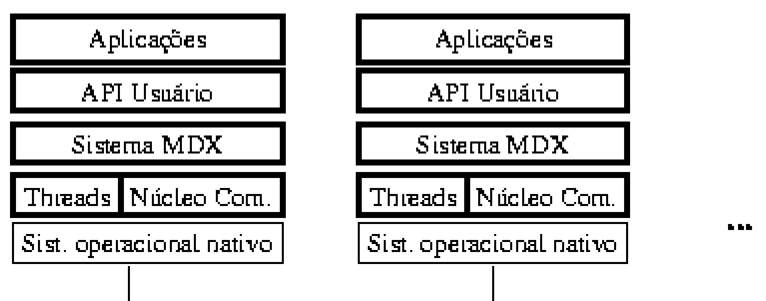

FIGURA 5.2 – Arquitetura básica do Sistema MDX sobre uma rede de trabalho.

5.1.3 Núcleo de comunicação

O núcleo de comunicação é um componente fundamental num ambiente de execução paralela e distribuída. Ele é implementado de acordo com um protocolo cliente-servidor, que permite a

independência de localização dos processos, interligando todos os clientes e todos os servidores de maneira transparente, rápida e confiável [PRE 98].

O núcleo de comunicação é compreendido por duas camadas: protocolo e encaminhamento de mensagens. A camada protocolo implementa um protocolo cliente-servidor de base. Todos os outros protocolos de comunicação do ambiente de programação paralela são implementados por servidores específicos. Esse protocolo mínimo é uma variante do protocolo RPC [BIR 84].

A camada encaminhamento de mensagens é composta por dois processos que fazem a recepção, o roteamento e o envio das mensagens, sendo um para as mensagens de servidores e outro para as mensagens dos clientes.

Nos sistemas tradicionais, o nível usuário solicita um serviço ao núcleo que o executa e retorna os resultados, por exemplo, utilizando os registradores da máquina. Na organização baseada em *microkernel* [COS 96], o núcleo de comunicação transmite somente o pedido e a resposta. O serviço é realizado por um servidor específico. Utilizando as vantagens da programação *multithread*, pode-se dividir as tarefas do núcleo em *threads*, permitindo que as mesmas executem em paralelo.

Funções do Núcleo de Comunicação

O núcleo de comunicação constitui a camada mais baixa do MDX. A função do núcleo é permitir a comunicação de todos os processos clientes com os processos servidores, que podem ser locais ou distantes, de maneira rápida, confiável e transparente, independente da localização dos mesmos [PRE 96a]. Ao núcleo de comunicação cabe as seguintes tarefas:

- examinar a mensagem e identificar se ela é uma requisição ou uma resposta e a quem ela se destina;
- localizar os processos clientes e servidores;
- enviar as mensagens.

Arquitetura do Núcleo de Comunicação

O núcleo de comunicação é formado por um processo que é executado em cada um dos nodos da rede. É composto por duas *threads* estáticas, várias *threads* criadas dinamicamente para garantir um alto grau de paralelismo, e por uma tabela de localização de nomes de servidores acessados pelo núcleo. As *threads* do núcleo de comunicação trocam mensagens locais entre si e com os processos clientes e servidores.

5.1.4 Servidores

O ambiente MDX é formado por servidores especializados mostrados na figura 5.1 [CCK 99]. O servidor ao receber uma mensagem executa um conjunto de serviços, podendo ou não devolver uma resposta para o cliente que solicitou. Para cada serviço requisitado, pode ser criada uma *thread* para a execução do mesmo, de modo a permitir a execução em paralelo.

Os servidores podem ser replicados ou centralizados. Um servidor replicado permite um melhor desempenho e a execução de serviços em mais de um nodo da rede. Por outro lado, um servidor centralizado garante a atomicidade para as operações.

Um servidor deve ser inicializado no ambiente MDX e ficar num laço infinito, recebendo requisições dos clientes, processando, preenchendo o cabeçalho de resposta e enviando-a para o cliente.

5.2 MDX-RS – Um Novo Núcleo de Comunicação MDX

Para fornecer comunicação entre autômatos RS distribuídos, o núcleo de comunicação MDX foi adaptado, originando um novo núcleo, denominado MDX-RS, via troca de mensagens [LIB 2000]. Na próxima seção, serão introduzidos alguns detalhes sobre o RS distribuído.

5.2.1 Necessidades do RS Distribuído

A comunicação entre autômatos RSD ocorre quando um autômato tem que enviar sinal a outro ou ao RS_Main. Pelo código fonte do arquivo de regras (arquivo *.rul*), quando um autômato tem a primitiva *emit(xxx)*, significa que ele vai emitir o sinal *xxx* para outro autômato, definido no arquivo de informações distribuídas (*.iod*). Por exemplo, no *mouse* distribuído (seção 3.3.2), um comando *emit(start)* envia o sinal *start* da máquina *sinope* para a máquina *pan*, porque no arquivo AID é indicado que *start* é um sinal de saída do autômato *mouseD1* (máquina *sinope*) e é um sinal de entrada para o autômato *mouseD2* (máquina *pan*).

Na tradução de um autômato RSD para um programa C [SCH 90], o comando *emit* é substituído pela primitiva de comunicação *MDX_send( )*, que envia um dado sinal a um determinado autômato. Para isso, já deve ter sido estabelecida uma conexão entre esses autômatos. Se a conexão não está estabelecida no momento do envio da mensagem, o núcleo se encarrega de fazer esta conexão com o autômato remoto.

Após ter sido estabelecida a conexão, cria-se um *socket* para a comunicação destes autômatos. E através deste *socket*, estes dois autômatos se comunicam até o fim da execução do programa RS distribuído.

Por exemplo, o comando *MDX_send( )*, para o caso acima descrito, ficaria deste modo:

```
MDX_send(mouseD2, "click");
```

onde *mouseD2* é o autômato que receberá a mensagem e *click* é o sinal enviado. A mensagem não está sendo enviada através do Protocolo RS, trata-se apenas de uma demonstração do *MDX_send( )*.

Para a recepção de um sinal, qualquer autômato tem uma *thread* que deve estar esperando o sinal em um comando *MDX_recv*. Os parâmetros deste comando são:

MDX_recv(int *Socket*, char *mensagem);

O primeiro argumento é o *socket* onde foi estabelecida a conexão entre os dois autômatos RSD, e a mensagem a ser recebida, que é posteriormente tratada de acordo com o Protocolo RS.

5.2.2 Funcionamento de um autômato

O autômato está sempre em estado de espera, aguardando um sinal de entrada para realizar uma computação.

Para exemplificar, serão considerados alguns trechos de código do mouse distribuído, que são executados na máquina *sinope*:

(Arquivo *mouseD1.aut*) 1 click [2, *, go_to(2)]

(Arquivo *mouseD1.rul*) 2. [ ] ===> [delta:=3, emit(start)]

O trecho (*mouseD1.aut*) acima quer dizer que, quando um sinal *click* chega do ambiente externo e o estado atual do sistema é "1", o RSD executa a regra "2" do arquivo de regras (*mouseD1.rul*) que atribui o valor "3" para a variável *delta*, emite *start* para o ambiente externo e passa o sistema para o estado "2". Neste caso, o novo estado do sistema será "2" e o sinal *start* terá sido emitido nesta reação.

O trecho abaixo indica que quando o sinal *tick* vem do ambiente externo e o estado atual do sistema é "1", o RSD simplesmente não executa nenhuma ação, pois a caixa de regras está vazia, e passa o sistema para o estado "1". Isso mostra que neste momento, o autômato não sofreu nenhuma alteração de estado, não emitiu nenhum sinal e nenhuma computação interna foi executada.

(Arquivo *mouseD1.aut*) 1 tick [3, *, go_to(1)]

(Arquivo *mouseD1.rul*) 3. [ ] ===> [ ]

Às vezes, o autômato ao receber um sinal, pode retornar outro, assim como pode somente fazer uma computação local, ou até, se o sinal for irrelevante naquele estado, não executar nenhuma operação.

O funcionamento de um autômato está esquematizado na figura 5.3:

FIGURA 5.3 – Funcionamento de um autômato RS distribuído

5.2.3 Estrutura do núcleo de comunicação MDX-RS

O modelo do antigo MDX é baseado em cliente/servidor, isto é, vários clientes na rede pedindo serviços prestados por um ou mais servidores. Como isso não se encaixa no modelo de autômatos distribuídos, pois um autômato não é exatamente nem um cliente, nem um servidor, foi definida uma nova configuração para o núcleo de comunicação do MDX, que difere daquela mostrada no capítulo 3.

5.2.3.1 NLT

O novo núcleo tem uma NLT (*Name Local Table*) onde estão cadastrados todos os nodos da rede que estão no sistema RS distribuído. Essa NLT é uma estrutura que tem como informações os seguintes campos:

TABELA 5.1 – NLT do novo núcleo MDX

Campo	Significado
char * sinal;	Sinais externos de entrada(*click*, *tick* ...)
unsigned int address;	IP da máquina
char * nome_aut;	nome do autômato RS (*mouseD1*)
unsigned int socket;	*socket* de envio/recebimento

5.2.3.2 Funcionamento do procedimento de estabelecimento de conexões

O procedimento *server_accept* do novo núcleo, é o responsável pelo estabelecimento das conexões entre os autômatos RS distribuídos, o RS_Main e o RS_IO. Ele recebe como parâmetro uma porta de comunicação, a qual é preestabelecida para a espera de novas conexões com outros autômatos. Então, o autômato atual fica esperando em um *accept* que outro autômato estabeleça uma conexão com ele. Quando essa conexão é estabelecida, as informações desta conexão vão sendo adicionado à NLT, e é gerada uma *thread* para fazer a conexão deste autômato remoto com o atual.

Esta *thread* é cíclica e só acaba quanto o autômato recebe uma mensagem pedindo a sua finalização. Dentro do ciclo, o autômato fica esperando uma mensagem através do *socket* criado no procedimento anterior. Ao receber uma mensagem qualquer, esta é tratada de acordo com o Protocolo RS (seção 4.3). Se após o tratamento da mensagem é necessário enviar sinal a um ou mais autômatos, então faz-se uma procura na NLT para encontrar os *sockets* em que os autômatos destinatários estão esperando e envia estas mensagens através de um Send( ). Este processo é esquematizado na figura 5.4.

FIGURA 5.4 – Processo de inicialização dos autômatos RS distribuídos

5.2.3.3 Esquema de funcionamento do novo núcleo

De posse das informações das seções anteriores, pode-se apresentar a estrutura de funcionamento do núcleo de comunicação do MDX, que provê suporte ao RS distribuído.

Sempre estará ativo um *Init( )* aguardando por novas conexões entre os autômatos RS distribuídos. A partir deste *Init( )*, a cada conexão estabelecida é criada uma *thread*

(procedimento *Recv( )*) que fica ativa para fazer a comunicação entre o autômato RSD local e o remoto. Essa nova estrutura está apresentada na figura 5.5:

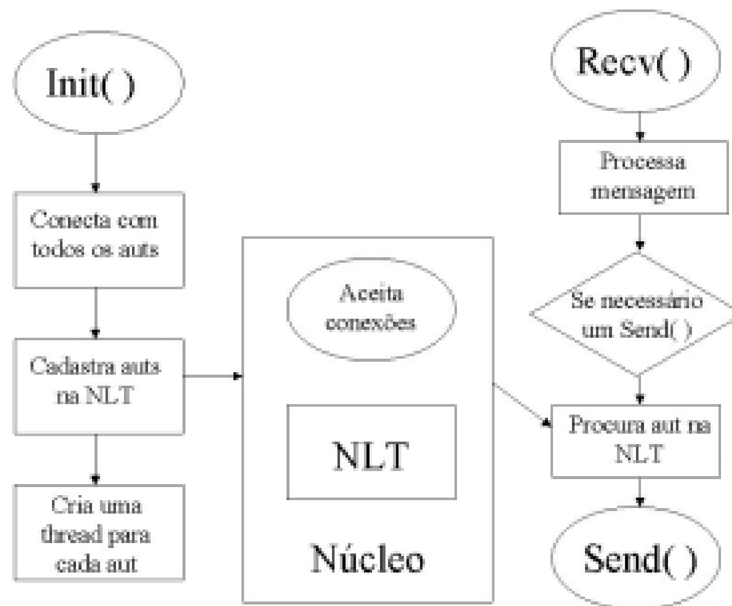

FIGURA 5.5 – Estrutura do novo núcleo de comunicação do MDX.

5.2.3.4 Nova estrutura do sistema MDX

A estrutura do antigo MDX tinha 5 etapas. São elas [COP 99]:

- **Aplicações**: Um programa C com as primitivas de comunicação e sincronização MDX, podendo ser clientes ou servidores, e podendo ter mais de uma aplicação em cada máquina.
- **API Usuário**: Faz a interface das aplicações com os servidores especializados. Ela dá a aparência de que as aplicações estão sendo realizadas em uma única máquina.
- **Servidores Especializados**: São os servidores espalhados pelo sistema MDX. Podem ser servidores de memória compartilhada, compilação, execução, sincronização e servidor de nomes.
- **Núcleo MDX**: É a parte principal do sistema MDX. Provém suporte de comunicação às aplicações, administra a NLT e localiza os servidores distribuídos, quando necessário.
- **Sistema Operacional**: Por enquanto, o sistema foi testado apenas sobre o Linux. Mas o projeto é que, futuramente, o MDX possa funcionar sobre qualquer sistema operacional.

Esse núcleo é representado na Figura 5.6:

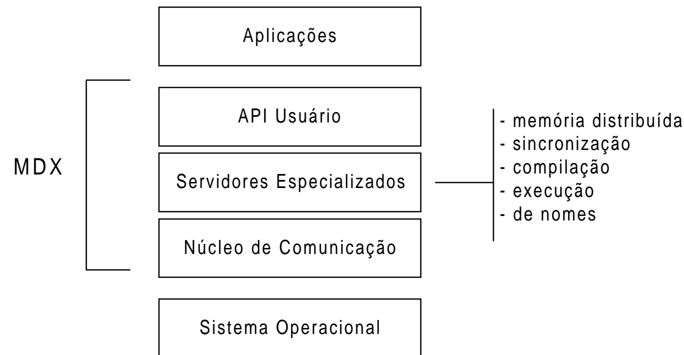

FIGURA 5.6 – Arquitetura do antigo sistema MDX

A nova estrutura do sistema MDX tem três etapas. São elas:

- **Autômato**: Um programa C com as primitivas de comunicação e sincronização MDX. O programa pode ser um autômato, ou o RS_Main e o RS_IO. Somente pode haver um autômato por máquina, exceto na máquina onde estão o RS_Main e RS_IO, a qual não pode ter autômato.
- **Novo Núcleo MDX**: É a parte principal do sistema. Provém suporte de comunicação às aplicações, administra a NLT e localiza os servidores distribuídos, quando necessário.
- **Sistema Operacional**: Por enquanto, também só foi testado sobre o Linux. Mas o projeto é que, futuramente, o novo núcleo do MDX possa funcionar sobre qualquer sistema operacional.

Esse núcleo é representado na Figura 5.7:

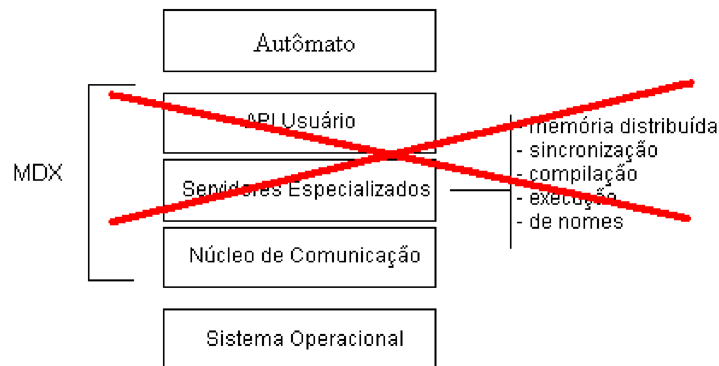

FIGURA 5.7 – Arquitetura do novo sistema MDX

A API do Usuário foi retirada do novo sistema MDX porque o próprio núcleo desenvolvido juntamente com a funcionalidade do modelo RSD, dão a transparência necessária (isto é, a aparência de que os autômatos estão executando em uma única máquina).

Os servidores especializados também não tem funcionalidade no RSD. O servidor de execução foi abolido, pois cada autômato tem que ser disparado manualmente em cada máquina do sistema, devido ao sistema MDX não ter ainda uma versão pronta deste servidor. O servidor de memória compartilhada não tem função, pois o sistema não utiliza compartilhamento de memória. A sincronização é feita pelos próprios RSDs, não se fazendo necessário um servidor. A compilação deve também ser feita manualmente pelo usuário, que com os códigos fontes, dispara cada autômato na máquina referida. E o servidor de nomes não existe mais porque cada autômato tem uma cópia da NLT em seu núcleo, onde estão indicados todos os autômatos que este nodo deve ter conhecimento.

A necessidade de um bom desempenho, é exigência do sistema RSD. Como o sistema suporta a hipótese do sincronismo para a obtenção dos autômatos (imaginando um processamento infinitamente veloz para os mesmos), então quanto mais rápida a conexão entre eles, melhor para o sistema RSD. Por isso, este novo núcleo de comunicação deve ter o melhor desempenho possível.

5.2.4 Comunicação no mouse distribuído

Nesta seção, é mostrado como é feita a conexão entre dois processos através do núcleo de comunicação do MDX, utilizando o exemplo do mouse distribuído. O sistema MDX é todo baseado em *sockets*; é através deles que são transmitidas e recebidas todas as mensagens do ambiente MDX [LIB 2000].

O núcleo MDX-RS é projetado para prover serviço de comunicação única e exclusivamente para autômatos RS distribuídos. Porém, pode-se usá-lo para prover comunicação a qualquer tipo de processo, desde que os mesmos tenham características semelhantes aos ARSD.

No sistema RSD, o número de autômatos presentes depende do programa e varia de uma aplicação para a outra. Esse número é somente é estipulado no próprio código fonte RSD. No caso geral, o núcleo deve fornecer conexão entre todos os autômatos do sistema. Como para cada conexão o autômato tem que estar a todo o momento esperando uma mensagem, foi decidido utilizar *threads*, onde cada *thread* fica responsável pela comunicação entre o autômato que a criou e uma outra *thread* remota. Cada *thread* se comunica com o autômato que a criou através de memória compartilhada local. Um autômato vai ter uma *thread* com cada outro autômato com o qual este tiver que trocar mensagens. Sendo *n* o número de autômatos do sistema, cada autômato terá no máximo *n-1 threads*. Isso sem contar a *thread* que faz a conexão com o RS_Main.

No exemplo do *mouse* distribuído, tem-se dois autômatos (*mouseD1* e *mouseD2*) que comunicam-se entre si, e ambos, obviamente, comunicam-se também com o *Mouse_Main*. Então, como os autômatos e o *Mouse_Main* estão em máquinas diferentes (somente o *Mouse_Main* e o *Mouse_IO* estão na mesma máquina e se comunicam através de memória compartilhada), a comunicação entre os mesmos se dá através de *sockets*, providos pelo núcleo de comunicação MDX-RS.

A escolha de máquinas diferentes para autômatos foi para simular um ambiente realmente distribuído, onde cada autômato RSD estaria em um *chip* ou controlador separado.

O *Mouse_Main* e o *Mouse_IO* estão na mesma máquina pois o *Mouse_IO* existe apenas no modelo de simulação para fazer uma interface com o usuário, via teclado e monitor.

Na Figura 5.8, a comunicação entre os autômatos através do núcleo MDX-RS é representada pelas setas pontilhadas. Note que essa comunicação somente é usada entre processos que estão em máquinas diferentes, nunca entre processos e/ou *threads* que estão na mesma máquina. Quando a máquina é a mesma, o meio de troca de mensagens é a memória compartilhada.

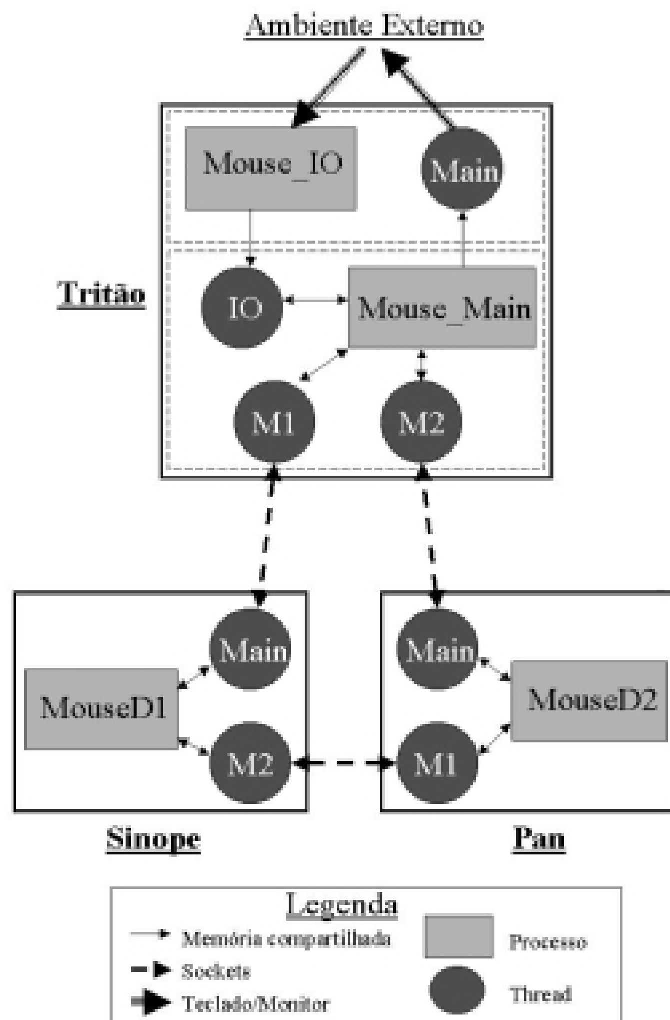

FIGURA 5.8 – Comunicação do *mouse* distribuído no novo sistema MDX

5.3 A escolha da linguagem C e do ambiente MDX

Como mencionado anteriormente, RS foi implementada em *SICStus Prolog*, um ambiente com bastante aceitação a nível acadêmico devido à segurança e à facilidades de programação oferecidas. Um ambiente com tais atributos certamente seria, a primeira vista, ideal para a distribuição da linguagem. De fato, dois fatores devem ser cuidadosamente analisados. O primeiro refere-se à confiabilidade já referida. Certamente, o *SICStus* é um ambiente bastante

seguro e um dos mais difundidos no âmbito acadêmico. Entretanto, a segurança e a difusão deste ambiente começam a diminuir quando fala-se em distribuição. Oferecendo pouquíssimas oportunidades de escolha, o *SICStus* é extremamente rígido nessa questão, impondo *sockets* como a única oportunidade de comunicação entre processos. Além disso, um ambiente *run-time* deve ser gerado utilizando-se um misto das linguagens C e Prolog, para possibilitar a execução do sistema sem a necessidade de inicializar o ambiente *SICStus*. O código gerado é lento e ineficiente.

O segundo fator a ser analisado é a programação. Como frisado no Capítulo 2, as linguagens reativas têm como objetivo a implementação da segunda camada de um ambiente – o núcleo reativo. As camadas de interface e de manipulação de dados podem, e geralmente são escritas em uma outra linguagem, também chamada de linguagem hospedeira. O Prolog não é indicado para ser usado como linguagem hospedeira. Tarefas simples e rotineiras tornam-se motivo de muita codificação. Por exemplo, a geração de um número randômico em Prolog pode-se tornar um problema.

A escolha da linguagem C baseia-se, principalmente, no último fator exposto. De fato, há muito desejava-se implementar RS em C para facilitar a programação das outras camadas. A linguagem C presta-se tanto para a implementação da camada de interface como para a manipulação de dados. A de manipulação de dados também se torna muito mais fácil, por ser uma linguagem "procedural" e possuir diversas bibliotecas prontas para uso na programação.

Escolhida a linguagem para ser usada como base para a distribuição, passou a ser necessária uma biblioteca que possibilitasse a distribuição. O fator mais importante na escolha do MDX foi seu modo de uso. Portando-se como uma biblioteca de comunicação do C, pode-se facilmente usá-la em qualquer plataforma em que ela tenha sido implementada. Outro fator relevante é o projeto de pesquisa envolvendo os grupos dos professores doutores Simão S. Toscani (UFRGS) e Celso Maciel da Costa (PUC-RS), onde estão em desenvolvimento o sistema RS e o ambiente MDX, respectivamente. Este trabalho seria um acréscimo no desenvolvimento, tanto da linguagem RS [TOS 93], quanto do ambiente MDX [PRE 98].

6 Exemplos de aplicações da Linguagem RS distribuída

Neste capítulo, são apresentados dois sistemas de controle: o primeiro mostra o controle de um veículo com inteligência própria, chamado de "*Três Olhos*" [COR XX]. Trata-se de um sistema centralizado cuja finalidade é apenas demonstrar a linguagem RS 5.0, isto é, que ela também pode ser usada no desenvolvimento de controles centralizados. O segundo exemplo, mostra um controle de tráfego terrestre que permite a diversos carros dirigirem-se a diferentes localidades, sem intervenção de um motorista ou de um operador. Visto a complexidade que possui este segundo exemplo, iremos nos concentrar em apenas um esboço da solução reativa do sistema. Nossa idéia é mostrar que a Linguagem RS 5.0 pode ser usada para diversos tipos de sistemas reativos, de forma bastante produtiva.

6.1 O Três Olhos

O objetivo desse exemplo é desenvolver um veículo completamente autônomo capaz de exibir um comportamento aparentemente inteligente. O Três Olhos (TÓ) deve vaguear, evitando obstáculos, e parar. Na situação de parado, fica atento ao aparecimento de qualquer objeto móvel com interesse. Caso isso aconteça, segue-o até o objeto parar por 3 segundos ou mais. Enquanto o objeto se move, o TÓ deverá seguí-lo, mantendo uma distância considerada confiável.

6.1.1 Arquitetura do TÓ

O material necessário para a implantação real desse sistema seria um carrinho com 2 motores: um para a direção (esquerda, direita, desligado) e outro para a locomoção (frente, parado, trás) e 3 sensores infravermelho (2 frontais e 1 traseiro, com saída analógica). A arquitetura de controle do veículo é modular e estruturada em 2 camadas, tal como na Figura 6.1.

Símbolos da Figura 6.1

(vertical)

Este símbolo indica que a saída pode representar tanto uma como a outra entrada. A saída é ativada quanto uma das entradas é ativa e desativada quando são ambas inativas. Deve-se garantir que as duas entradas nunca estão inativas simultaneamente, o que neste sistema é feito através das ligações entre os módulos VAGUEAR, CLOCK e SEGUIR.

(horizontal)

Este símbolo tem um significado semelhante ao anterior, com a condicionante de a entrada direcionada ser prioritária relativamente a outra.

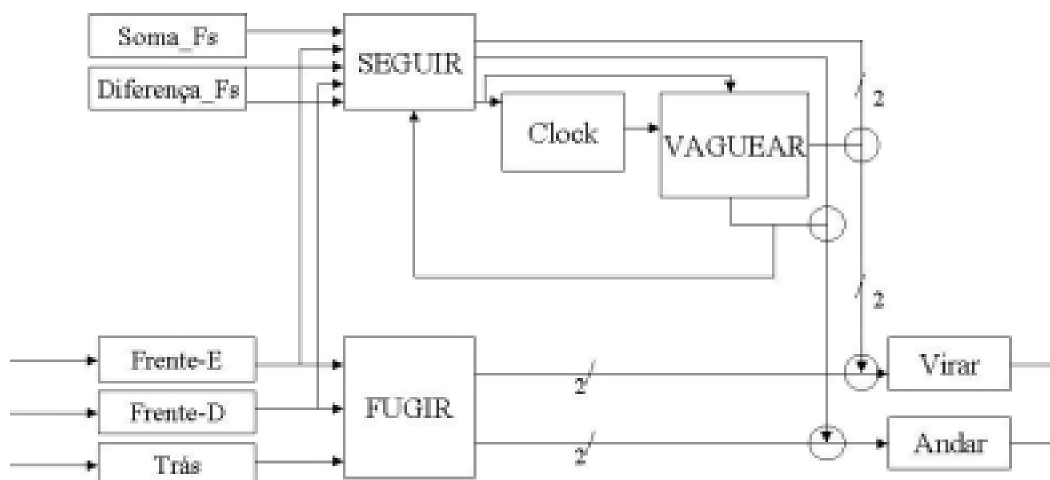

FIGURA 6.1 – Arquitetura do Controle do Três Olhos

6.1.2 Descrição dos Módulos

Os módulos Virar e Andar correspondem aos “drivers” dos motores do TÓ. O módulo CLOCK gera impulsos periódicos. Este ainda uma entrada (saída de SEGUIR) que enquanto ativa, inibe e reinicializa o funcionamento deste módulo.

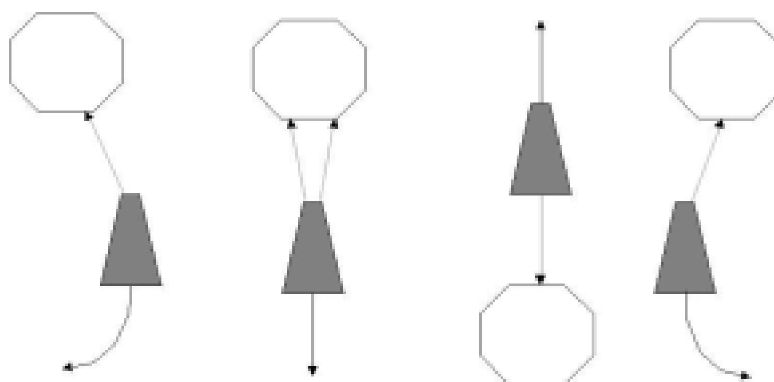

FIGURA 6.2 – Reações do TÓ a um obstáculo (módulo FUGIR)

O primeiro nível de controle da arquitetura corresponde apenas ao módulo FUGIR. Este módulo, sempre ativo, tem como tarefa evitar que o veículo colida com obstáculos ou objetos móveis. A especificação do módulo encontra-se na Figura 6.2, com a indicação da

reação desejada do TÓ para as 4 situações representadas. Das situações não representadas, se exige apenas que o TÓ fique parado, enquanto os três sensores estiverem ativos.

O módulo VAGUEAR é responsável por uma primeira parte do segundo nível de controle. Pretende-se com esse comportamento que o TÓ ande, apenas para frente, e pare aleatoriamente, com uma probabilidade de 25% para andar, e o restante, para que fique parado. Enquanto está andando, deve fazê-lo normalmente em linha reta – com 50% de probabilidade – e, aleatoriamente, virar à esquerda ou à direita – com 25% de probabilidade para cada lado. A modificação do estado andar/parado deve ser feita com um período de 3 segundos. A mudança de direção, que só pode acontecer se o veículo estiver em movimento, e deve ter um período de 1 segundo. Deve-se notar que o TÓ não deve gastar todo o período de 1 segundo para o giro da sua direção, pois caso isso aconteça, originaria ângulos muito fechados de direção – por isso, aconselha-se 0,5 segundos. Existe ainda uma entrada Reinicialização que coloca este módulo em um estado inicial. Esta entrada é ativada pelo módulo SEGUIR, agindo simultaneamente no módulo CLOCK, e inibe o comportamento de VAGUEAR enquanto o TÓ segue um objeto. Por outro lado, a saída de VAGUEAR, para o motor de movimento, vai inibir o comportamento de SEGUIR, conseguindo-se assim, exclusão mútua entre estes dois comportamentos.

O módulo SEGUIR só pode ser ativado quando o TÓ se encontra parado. Este comportamento pode ser decomposto em duas máquinas de estado, uma que faz mover o veículo e outra que o deve manter centrado com o objeto a SEGUIR.

- Máquina de comportamento: quando um objeto se coloca a frente do TÓ, a uma distância inferior a Soma-Fs, isto é, ativando o sinal Soma-Fs (ver figura 6.3), torna-se num objeto com interesse para ser seguido. Nessa situação, o SEGUIR deve fazer o TÓ andar para frente até ativar um dos sensores frontais. Quando isso acontecer, o SEGUIR deve parar o veículo. A partir dessa seqüência, o SEGUIR deve fazer o TÓ andar (para a frente) quando o objeto deixar de ativar o sinal Soma-Fs, e pará-lo logo que Soma-Fs voltar a ficar ativo. Só atua no bit que permite o veículo andar para a frente. Se o objeto ficar parado por 3 segundos ou mais, iniciar o módulo VAGUEAR.

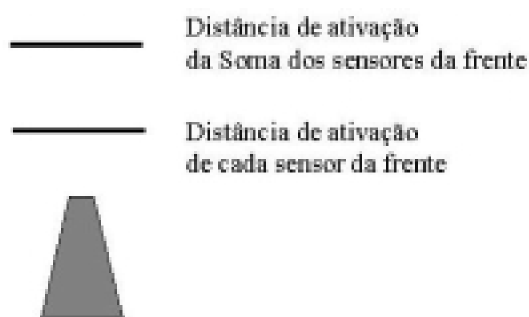

FIGURA 6.3 – Limites das distâncias que o Três Olhos é sensível.

- Máquina de centragem: é responsável por manter o veículo centrado (na perpendicular) com o objeto a ser seguido. Enquanto dois bits de Diferença-Fs indicarem que o objeto está a uma distância idêntica dos dois sensores frontais (D1 = D2), o SEGUIR deve manter o TÓ a andar em linha reta. Quando isso não acontecer, deve-se virar para o lado em que o sinal é menor. Esta máquina de centragem só deve funcionar caso a máquina de

movimento esteja responsável pelo movimento do TÓ. Ela atua nos 2 bits do motor de direção.

O módulo SEGUIR gera ainda um sinal A_SEGUIR (Reinicializa), indicando que se encontra a SEGUIR um objeto. Este sinal, quando ativo, vai inibir os módulos CLOCK e VAGUEAR.

Deve-se notar que deve ser possível testar o TÓ com apenas o primeiro nível de controle, ou com este e qualquer um dos comportamentos do nível superior (VAGUEAR e SEGUIR), demonstrando-se a modularidade do sistema.

6.2.3 Implementação em RS

- **Modulo *controla_motores*:**

O módulo *controla_motores* tem como função controlar os dois motores do TÓ, isto é, tanto o motor de locomoção, quanto o motor de direção.

1. Sinais de Entrada:
   - *Ir_frente(X)*: se X=0, o TÓ vai andar para frente. Se X=1, ele ficará parado e se X=2, o TÓ andará para trás;
   - *Ir_lado(Y)*: Se Y=0, o TÓ virará à esquerda. Se Y=1, a direção ficará centralizada, e se Y=2, a direção será dobrada à direita;
   - *Fuga_tras_esq*: O TÓ fugirá de um obstáculo indo para trás e à esquerda;
   - *Fuga_tras_dir*: O TÓ fugirá de um obstáculo indo para trás e à direita;
   - *Fuga_frente_desl*: O TÓ fugirá de um obstáculo indo para frente e reto;
   - *Tick*: um milissegundo.

2. Sinais de Saída:
   - *Andar(X)*: aciona os motores do TÓ para a locomoção;
   - *Virar(Y)*: aciona os motores do TÓ para a direção.

- **Módulo *fugir*:**

Este módulo tem como função exclusiva receber os sinais dos sensores, indicando que existem objetos próximos de si, e fazer com que o TÓ fuja deles.

1. Sinais de Entrada:
   - *E*: sinal do sensor à esquerda;
   - *D*: sinal do sensor à direita;
   - *T*: sinal do sensor da parte de trás do TÓ;

2. Sinais de Saída:
   - *Fuga_tras_esq*: O TÓ fugirá de um obstáculo indo para trás e à direita;
   - *Fuga_tras_dir*: O TÓ fugirá de um obstáculo indo para trás e à direita;
   - *Fuga_frente_desl*: O TÓ fugirá de um obstáculo indo para frente e reto;

- **Modulo *vaguear*:**

Este módulo controla o TÓ quando ele não está nem fugindo de algum obstáculo, nem seguindo algum objeto. Isto é, quando o TÓ não tem o que fazer, fica vagando pelo plano.

1. Sinais de Entrada:
   - *Tick*: um milisegundo;
   - *Reinicializa*: sinal recebido para iniciar o processo de vaguear do TÓ.

2. Sinais de Saída:
   - *Ir_frente(X)*: se X=0, o TÓ vai andar para frente. Se X=1, ele ficará parado e se X=2, o TÓ andará para trás;
   - *Ir_lado(Y)*: Se Y=0, o TÓ virará à esquerda. Se Y=1, a direção ficará centralizada, e se Y=2, a direção será dobrada à direita;
   - *Desinibe_seguir*: verifica a existência de um objeto interessante a ser seguido, e dispara o módulo *to_seg*.

- **Modulo *to_seg*:**

É o módulo responsável pelo controle do TÓ quando este encontra um objeto interessante de ser seguido. Este módulo controla-o enquanto o objeto estiver à uma distância segura.

1. Sinais de Entrada:
   - *E*: sinal do sensor à esquerda;
   - *D*: sinal do sensor à direita;
   - *T*: sinal do sensor da parte de trás do TÓ;
   - *OE*: sinal para seguir um objeto à frente e à esquerda;
   - *OD*: sinal para seguir um objeto à frente e à direita;
   - *L*: sinal para parar de seguir o objeto;
   - *Tick*: um milisegundo;
   - *Desinibe_seguir*: verifica a existência de um objeto interessante a ser seguido, e dispara o módulo *to_seg*.

2. Sinais de Saída:
   - *Ir_frente(X)*: se X=0, o TÓ vai andar para frente. Se X=1, ele ficará parado e se X=2, o TÓ andará para trás;
   - *Ir_lado(Y)*: Se Y=0, o TÓ virará à esquerda. Se Y=1, a direção ficará centralizada, e se Y=2, a direção será dobrada à direita;
   - *Reinicializa*: sinal recebido para iniciar o processo de vaguear do TÓ.

```
rs_prog to:
 input  : e, d, t, oe, od, l, tick, reinicializa;
 output : andar(X), virar(Y);

 module controla_motores:
   input  :  ir_frente(X), ir_lado(Y), fuga_tras_esq, fuga_tras_dir, fuga_frente_desl, tick;
   output :  andar(X), virar(Y);
   p_signal: passo0, passo1;
   var     : cm_frente, cm_lado, cm_temp;
   initially : up(passo0);
   on_exception :
```

```
            .......
    end on_exception;
    ir_frente(X)+[passo0] ==> cm_frente:=X, emit(andar(cm_frente)), up(passo0);
    ir_lado(Y)+[passo0] ==> cm_lado:=Y, up(passo1);
   /* assim com outros sinais */
end module;

  module fugir:
    input  : e,d,t;
    output : fuga_tras_esq, fuga_tras_dir, fuga_frente_desl;
    e ==> emit(fuga_tras_esq);
    d ==> emit(fuga_tras_dir);
    t ==> emit(fuga_frente_desl);
  end module;

  module vaguear:
    input  : tick, reinicializa;
    output : ir_frente(X), ir_lado(Y);
    t_signal: next1, next2;
    var     : v_count, v_tmp, v_tmp1;
    initially : v_count:=0, activate(decisao), activate(timer);
    box decisao :
       reinicializa ==> random(N), v_tmp1:=N, up(next1);
       +[next1]  ==> case
               v_tmp1==1 --> random(K), v_tmp:=K, up(next2), v_count:=0, activate(timer);
              else --> /*emit(desinibe_seguir),*/ v_tmp:=1;
         end case;
     end box;
   /* assim com outros sinais e caixas */
  end module;

  module to_seg:
    input  : e,d,t,oe,od,l, tick, desinibe_seguir;
    output : ir_frente(X), ir_lado(Y), reinicializa;
    t_signal: segue_esq, segue_dir, segue_reto, para;
    p_signal: se, sd;
    var     : count, segtmp;
    initially : activate(centrar), activate(espera);
     on_exception :
          excpt  ==> count:=0, activate(centrar), activate(espera);
     end on_exception;
     box espera:
      desinibe_seguir ==> count:=0, exit_to(state_1);
     end box;
      /* assim com outros sinais e caixas */
   end module;
end rs_prog.
```

6.2 Auto-estrada do Futuro

O objetivo deste exemplo é esboçar um sistema para o controle de uma auto-estrada, onde os carros são controlados através de receptores instalados nos mesmos, que recebem sinais de uma central que utiliza sensores colocados em postes situados ao lado da estrada. Os carros seguem em comboios, em uma velocidade constante sem a necessidade de um motorista.

6.2.1 Descrição

O sistema aqui descrito tem uma tarefa básica: o controle de todos os automóveis que se encontram na pista, fazendo-os virar, acelerar ou frear no momento correto. Deve, ainda, decidir qual o melhor caminho para ser percorrido. A central tem um registro de todos os destinos (informados no início da viagem) e a posição onde cada carro se encontra.

É definido que a pista deve conter sensores em toda a sua extensão. Tais sensores emitem uma mensagem ao automóvel que está passando, avisando-o do local onde este se encontra (o identificador daquele sensor). Os sensores calculam a velocidade do veículo no momento em que ele passa pelo sensor. O cálculo é feito com o auxílio de dois sensores infravermelhos. Quando o primeiro sensor é acionado (pela passagem de um carro), o cronômetro é zerado e fica ativo até que o segundo sensor seja ativado. Como a distância dos sensores é fixa, o tempo é o único fator necessário para o cálculo da velocidade.

O carro deve responder com um sinal de *acknowledgment* a todas as mensagens que recebe. Desta maneira, garante-se que as mensagens sejam entendidas corretamente. Além disso, o veículo possui um *self-test*, responsável pela verificação de todos os seus sistemas – pastilhas de freio, quantidade de gasolina, funcionamento do motor, etc. Assim, qualquer falha pode ser avisada à central, que indicará as providências cabíveis a serem tomadas.

Nessa estrada, todos os carros tendem a andar em comboios. Como os carros vão entrando aleatoriamente na estrada, a central decide quais os carros que devem esperar para a formação de um novo comboio e quais os carros que manterão a velocidade máxima para completar o comboio à frente.

Uma particularidade são as emergências que não podem ser tratadas pela central. A mais clara é relativa à falha da central, que deixaria de enviar sinais ou os enviaria, mas de forma incompreensível. Neste exemplo, os veículos simplesmente param nesta situação. Como a ausência de sinal ou o não-entendimento deste pode ser um fator local (uma pane no veículo), o veículo deve emitir uma mensagem a central avisando sua iniciativa de parar. Se a central não estiver com problemas e for apenas um problema local no veículo, ela tomara as providências necessárias. Enquanto isso, os veículos atrás daquele identificariam o incidente através de um sonar com o qual todos os veículos estão equipados. Assim, impede-se que um veículo cause um acidente com os carros que se encontram atrás ou que ande de forma desgovernada no caso de uma queda ou mal funcionamento da central.

Outra situação é o surgimento de algum obstáculo na pista (desmoronamento, por exemplo) que impeça que o carro prossiga normalmente seu rumo. Como tudo pode ocorrer muito rapidamente, é provável que os sensores na pista não detectem o problema a tempo (ou que tenham desmoronado junto com a pista). Nesse caso, o motorista teria à sua disposição um botão, que iniciaria uma freada brusca. O veículo deve, ao mesmo tempo, avisar à central a respeito incidente. O modo com que os outros veículos ficam sabendo do incidente é pela central, que reenviaria a mensagem a todos os veículos que podem ser lesados e, novamente, pelo sonar. Isso se faz necessário a partir do momento em que a comunicação com a central seja feita via satélite, que pode produzir um atraso relativo do sinal. Como a tendência nessas pistas automáticas é que todos os carros trafeguem separados por distâncias curtas e em alta velocidade, um dispositivo como o sonar, que se utiliza de microondas e produz respostas

bem mais rápidas do que comunicação via-satélite, tem melhores condições de evitar um possível desastre.

Embora exista um controle bastante distribuído, a tarefa mais complexa ainda fica por conta da central. Ela se encarrega de informar a todos os carros quando devem virar, acelerar ou frear; recebe mensagens da pista e dos carros, que devem ser processadas e retornadas ao carro da maneira mais rápida possível; embora isso não tenha sido implementado no nosso exemplo, uma central deve indicar que um veículo não é mais de sua responsabilidade, passando-o a outra central. As mensagens possuem um tempo fixo que deve ser rigidamente respeitado. Por exemplo, as mensagens *acelerar* e *virar* devem ser emitidas a todos os carros de *y* em *y* segundos. Quando passam *2y* segundos sem mensagens, os veículos podem presumir que uma falha ocorreu e que o problema deve ser tratado.

6.2.2 Descrição dos Módulos

Os carros:

O sistema de controle dos carros é descrito da seguinte forma: assim que o carro chegar a beira da auto-estrada, ele avisará à central que está aguardando um sinal para entrar na via. Os sensores checarão o trânsito no local, e assim que não houver perigo, dará o sinal para o carro entrar. Logo após, será recebido dos sensores de controle espalhados pela estrada, um sinal informando o quilômetro (km) atual, em que ele se encontra ao entrar na auto-estrada. Também como sinal de entrada, é recebido a velocidade que o carro deverá manter constante na via.

Os carros terão um módulo que fará uma verificação da sua distância em relação ao veículo à sua frente e ao veículo localizado à sua traseira. Essas distâncias serão obtidas pelo sonar, um localizado na frente e o outro na parte de trás do carro. Esse módulo fará uma checagem de 0,1 em 0,1 segundos, para verificar se uma manobra de emergência deve ser tomada. Isso pode acontecer quando, por exemplo, o carro à sua frente perde velocidade subitamente, ou o carro que está atrás vem em alta velocidade.

Toda vez que um carro entrar na estrada, os sensores verificarão se algum outro veículo está próximo desse entroncamento. Se algum carro estiver nas mediações, estes receberão um aviso para diminuir a sua velocidade. Assim que o primeiro carro estiver em ritmo constante, todos os carros que tiveram sua velocidade reduzida, receberão indicação para retomarem a sua velocidade normal.

O próprio carro controla em qual das pistas ele se encontra. E sempre que for necessária uma mudança de pista, será feita uma requisição à central de controle. Essas, farão uma averiguação para ver se essa manobra pode ser feita com segurança, ou seja, se não tem nenhum carro na outra pista, e se há uma distância suficiente para a troca de faixa. Se a troca pode ser feita, a central informa o carro e este faz a manobra solicitada.

Os sensores

Os sensores tem como função indicar, aos carros e à central, a real localização dos automóveis e a velocidade atual dos mesmos. Eles ainda fazem a verificação da pista, indicando se ela está liberada para tráfego, isto é, sem algum obstáculo (carro, árvore, etc.) impedindo o fluxo de veículos. Os sensores de comunicação estão espalhados por toda a auto-estrada, com uma distância de X metros entre elas.

A central de controle

A central de controle é responsável por todo o controle da pista. Ela verifica a localização dos carros, recebe as solicitações dos carros, e de acordo com a viabilidade, autoriza-os ou não. Tem a responsabilidade de formar os comboios. Decide quais os carros que devem aguardar o comboio e quais os carros que devem manter a velocidade máxima permitida para a formação dos mesmos.

A central também tem a função de decidir a velocidade limite em cada trecho da pista. Ela utilizará sensores que indicarão as condições climáticas da estrada a cada momento, para proporcionar uma velocidade limite maior ou menor, a cada trecho, a todos os carros.

6.2.3 Implementação em RS

Conforme já mencionado, o exemplo apresentado nesta seção tem como única finalidade exemplificar a nova linguagem e sua potencialidade. O problema sequer está completamente especificado. Sistemas como esse são considerados estado-da-arte e possuem uma implementação extensa, dado o número de controles e reações que devem ser tratadas. Iremos concentrar-nos apenas na resolução reativa do sistema. Para isso, todos os sinais devem ser identificados. Assim, dividiremos nosso sistema em três autômatos distintos que controlam, respectivamente, a central, os veículos e os sensores nas pistas.

- **A central**

1. Sinais de Entrada:
   - *Pos(Veiculo_ID, X)*: indica a posição em que um determinado automóvel se encontra. O parâmetro *X* é o identificador do sensor. Pode-se, a partir dele, saber a exata posição do automóvel;
   - *Velocidade(Veiculo_ID, X)*: indica a que velocidade o carro está se movendo. *X* é a velocidade em Km/h;
   - *Itinerário(Veiculo_ID, X)*: define ou redefine o destino desejado por um determinado veículo. *X* é o destino desejado pelo veículo;
   - *Central_tick*: emitido a cada milésimo de segundo;
   - *Pane(Veiculo_ID, X)*: sinal que indica que um determinado automóvel encontra-se em um estado de mal-funcionamento. O parâmetro *X* indica qual é exatamente o tipo de pane apresentada;
   - *Requerer_autorizacao(X)*: o carro X pede para entrar na pista;

- *Carro_na_estrada(X)*: confirmação que o carro X entrou na pista;
- *Trocar_pista(Veiculo_ID, Q):* veículo solicita trocar sua faixa atual;
- *Clima(K)*: recebe informação do clima no momento atual, para o controle de velocidade máxima.

2. Sinais de Saída:
   - *Virar(Veiculo_ID, X)*: indica que um determinado carro deve virar *X* graus;
   - *Andar(Veiculo_ID, X)*: indica que um determinado carro deve ter uma velocidade de *X* quilômetros por hora. Se, no momento em que ele receber a mensagem, estiver mais rápido, deverá frear; se estiver mais lento, deverá acelerar; se estiver na mesma velocidade, a mensagem não terá efeito algum;
   - *Emergência(X)*: sinal que indica uma determinada situação de emergência. Emitida a todos os veículos;
   - *Operador(X)*: este sinal tem a função de avisar ao operador da central o tipo de pane que está ocorrendo para que este possa intervir. O parâmetro *X* indica o tipo de falha ou pane;
   - *Autorizacao(veiculo_ID)*: veículo recebe a autorização para trafegar na pista;
   - *Carro_entrou(veiculo_ID)*: avisar veículos que outro entrou logo à sua frente.

- **Os sensores da pista**

1. Sinais de Entrada:
   - *SE(Veiculo_ID)*: sensor esquerdo foi acionado pelo veículo com a identificação determinada em seu parâmetro;
   - *SD(Veiculo_ID)*: sensor direito foi acionado;
   - *ACK(MsgID)*: indica que a mensagem informada pelo parâmetro foi corretamente recebida e compreendida;
   - *Sensor_tick*: emitido a cada milésimo de segundo.

2. Sinais de Saída:
   - *Sensor_ID(X)*: mensagem enviada aos veículos indicando o identificador do sensor. A partir disso, pode-se saber em que posição o automóvel se encontra;
   - *Velocidade(Veiculo_ID, X)*: mensagem enviada à central informando a velocidade do veículo
   - *Pos(Veiculo_ID, X)*: mensagem informando a posição do veículo;

- **O veículo**

1. Sinais de Entrada:
   - *Virar(Veiculo_ID, X)*: indica que o veículo (se o parâmetro *Veiculo_ID* corresponder ao seu ID) deve virar *X* graus;
   - *Andar(Veiculo_ID, X)*: indica que o veículo (se o parâmetro *Veiculo_ID* corresponder ao seu ID) deve ter uma velocidade de *X* quilômetros por hora. Se, no momento que ele receber a mensagem, estiver mais rápido, deverá frear; se estiver mais lento, deverá acelerar; se estiver na mesma velocidade, nada deverá ser feito;
   - *Veiculo_tick*: emitido a cada milésimo de segundo;
   - *Sonar(X)*: sinal do sonar do carro;

- *Sensor_ID(X)*: mensagem enviada aos veículos indicando o identificador do sensor. A partir desse dado, pode-se determinar onde o automóvel se encontra;
- A*utorizacao(veiculo_ID)*: autorização recebida para entrar na pista;
- *Carro_entrou(veiculo_ID)*: indica que um outro veículo entrou logo à sua frente;

2. Sinais de Saída:
   - *ACK(MsgID)*: indica que a mensagem informada pelo parâmetro foi corretamente recebida e compreendida;
   - *Itinerário(Veiculo_ID, X)*: define ou redefine (o sinal pode ser emitido mais de uma vez) o destino desejado pelo veículo;
   - *Pane(Veiculo_ID, X)*: indica que o automóvel encontra-se em um estado de mal-funcionamento. O parâmetro *X* indica qual é exatamente o tipo de pane ocorrida;
   - *Parando(Posicao)*: sinal que indica que por algum tipo de emergência ou mal funcionamento da central, o veículo está parando. O parâmetro do sinal, *Posição*, indica qual o último sensor por que o veículo passou;
   - *Requerer_autorizacao*: solicitação de autorização. O carro se encontra a margem da pista e pediu autorização à central para trafegar;
   - *Carro_na_Estrada(Veiculo_ID)*: após solicitar autorização, o carro avisa que entrou na auto-estrada;
   - *Trocar_pista(Veiculo_ID, Q)*: permissão para o carro, que se encontra na faixa Q, trocar de faixa, pois uma ultrapassagem está para ser feita.

De posse dessas informações, pode-se partir para a programação propriamente dita. O código RSD é apresentado abaixo. Os sinais externos de entrada são definidos de forma a possibilitar uma simulação do sistema no ambiente RSD. Por outro lado, o único sinal de saída - *Operador(X)* - tem a real função de avisar ao operador o problema que está ocorrendo.

```
rsd_prog controle_trafego:
external interface:
  input: se(Veiculo_ID), sd(Veiculo_ID), central_tick, sensor_tick, veiculo_tick,
         itinerario(Veiculo_ID, X), pane(Veiculo_ID, X);
  output: operador(X);
end interface;

machine central:
input: pos(Veiculo_ID, X), velocidade(Veiculo_ID, X), itinerario(Veiculo_ID, X), pane(Veiculo_ID, X),
       central_tick, parando(Posicao);
  output: andar(Veiculo_ID, X), virar(Veiculo_ID, X), emergencia(X),  operador(X);
  velocidade(Veiculo_ID,X) ⇒ /* guarda velocidade para tomar decisão abaixo */
  pos(Veiculo_ID, X) ⇒ case
                  /* decide qual o proximo movimento deste carro */ → emit(andar(Veiculo_ID, y)),
                  emit(virar(Veiculo_ID, z));
          end case;
  /* Supõe-se, aqui, que a pane número 2 deve ser avisada ao operador. */
  pane(Veiculo_ID, X) ⇒ case
                          X = 2 → emit(operador(X));
                          else → /* outra atitude */
                      end case;
/* Todos os outros sinais são tratados de forma semelhante */
end machine;

/* Nesse exemplo, o identificador do sensor é 3 */
machine sensor:
  input: se(Veiculo_ID), sd(Veiculo_ID), sensor_tick, ack(Msg_ID);
  output: sensor_ID(X), pos(Veiculo_ID, X), velocidade(Veiculo_ID, X);
  t_signal: carro_passando;
```

```
  var: tempo, veiculo;
  initially: delta := 0;
  se(Veiculo_ID) ⇒ up(carro_passando), tempo := 0, veiculo := Veiculo_ID,
                              emit(sensor_ID(3), pos(Veiculo_ID, 3));
  sd(Veiculo_ID) ⇒ /* calcula velocidade */, emit(velocidade(veiculo, z);
  sensor_tick+[carro_passando] ⇒ tempo := tempo + 1, up(carro_passando);
  /* assim com outros sinais */
end machine;

/* Nesse exemplo, o carro possui um ID = 5 */
machine car:
 input: virar(Veiculo_ID,X), andar(Veiculo_ID,X), veiculo_tick, parando(posicao), sensor_ID(X), sonar(X);
 output: ack(Msg_ID), itinerario(Veiculo_ID, X), pane(Veiculo_ID, X);
 var: ult_veiculo;   /* indica o ID do carro referente a última mensagem */
 virar(Veiculo_ID, X) ⇒ case  Veiculo_ID = 5 → /*Vira o veículo X graus*/;
                                    else → ult_veiculo := Veiculo_ID;
                              end case;
  andar(Veiculo_ID, X) ⇒ case Veiculo_ID=5 → /*Veloc. passa para X Km/h*/;
                                          else → ult_veiculo := Veiculo_ID;
                                      end case;
  sensor_ID(X) ⇒ emit(ack(y));
  /* assim com os outros sinais são tratados de forma semelhante */
end machine;
end rsd_prog.
```

Este esboço de solução, embora bastante incompleto (como não poderia deixar de ser), mostra alguns fundamentos da nova linguagem, bem como os recursos oferecidos ao programador. Todos os sinais que forem emitidos por um ARSD serão automaticamente enviados para todos outros que possuam aquele sinal como um sinal de entrada válido. Como se pode perceber, a programação reativa não se torna exageradamente complexa.

7 Conclusão

Este trabalho envolveu quatro tarefas principais: a definição de uma nova versão da Linguagem RS, versão essa que permite a distribuição de autômatos, a criação de um novo compilador, onde as mudanças na linguagem foram implementadas, a implementação de um ambiente de execução baseado no sistema MDX e a demonstração do uso da linguagem.

A nova versão da linguagem, denominada RS 5.0, é uma extensão da versão RS 4.0 [TOS 93] que além de ampliar a capacidade da linguagem, melhorou a sintaxe de seus programas, facilitando a programação dos mesmos. A linguagem permite o uso de mais de um sinal externo por condição de disparo, bem como o uso de sinais inibidores, deixando de ser necessário o uso de declarações vazias e apresentando relações de exclusão mútua e de concomitância entre sinais externos.

Essas alterações facilitam a programação de sistemas reativos, pois os programas passam a poder usar mais de um sinal no gatilho de disparo de uma regra e esses sinais podem chegar do ambiente ao mesmo tempo, algo que não era possível na versão anterior da linguagem. Outra novidade é o uso de sinais inibidores, que evitam a execução de regras quando estão presentes.

Para ter-se um melhor controle sobre os sinais que podem chegar do ambiente externo ao mesmo tempo, foram criadas duas regras, já existentes em outras linguagens, como Esterel. Essas regras são as de exclusão mútua e de concomitância. A primeira, declara que determinados sinais jamais chegarão juntos do ambiente externo. A regra de concomitância, diz que um sinal pode automaticamente indica a presença de outro. Isto é, toda a vez que um sinal for disparado, implicará na presença de outro sinal de entrada.

A distribuição da linguagem envolveu, basicamente, três tarefas: a definição de um modelo genérico que possibilitasse a distribuição da linguagem, a definição e a implementação de um modelo para um núcleo de comunicação MDX-RS, para a troca de sinais entre ARSD, e a implementação de um novo compilador.

A definição do modelo do RSD atingiu a linguagem nos seguintes pontos:

- Mudanças na sintaxe introduziram o novo comando *machine* permitindo que, com apenas um código fonte, produza-se um programa que execute em diversas máquinas.
- Mudanças na semântica do comando *emit*, que antes significava a emissão de um sinal para o ambiente externo, agora possui também a função de estimular outros autômatos enviando-lhes mensagens que lhes sejam relevantes.
- A inclusão de um protocolo de comunicação para os autômatos distribuídos entre as máquinas, permitiu o correto funcionamento do sistema. O protocolo utiliza o algoritmo de Lamport para a ordenação das mensagens.

O modelo definiu dois processos que sempre são gerados: o RS_IO e o RS_Main. O primeiro tem como tarefa receber os sinais digitados pelo usuário e repassá-los ao RS_Main.

O segundo tem a função de receber sinais proveniente do RS_IO e repassá-los aos autômatos, e gerenciar outros controles, tais como a finalização do sistema, casos de pane, inicialização dos ARSDs, etc. RS_IO só existe em tempo de depuração, podendo ser descartado quando da instalação real do sistema, momento em que os sinais serão enviados diretamente ao RS_Main.

O núcleo de comunicação baseia-se na utilização de *sockets* e teve como objetivo melhorar o desempenho do núcleo de comunicação existente no MDX. O núcleo original provê uma abstração de memória compartilhada entre os processos do sistema, o que compromete o seu desempenho. De acordo com vários estudos, as bibliotecas de comunicação baseadas em memória virtual compartilhada tem desempenho inferior às bibliotecas de comunicação baseadas em troca de mensagens [PAR 97] [FEL 95].

Para efetivar todas as inclusões e alterações na linguagem RS, foi desenvolvido um novo compilador. A implementação deste novo compilador foi realizada em C e resultou num protótipo que inclui facilidades para depuração de programas.

O Compilador RS 5.0 é implementado em C-ANSI e, por conseqüência, pode ser facilmente executado em diversos ambientes. Praticamente todos os sistemas, como por exemplo, PC, Macintosh, Amiga, etc., possuem um compilador que aceita comandos C-ANSI.

O Compilador RS 5.0 gera como saída três tipos de códigos. O primeiro tipo é o autômato padrão RS, o qual era também gerado pela versão RS 4.0. Este código é gerado em dois arquivos, onde um contém a descrição do autômato e outro as regras referenciadas por este.

O segundo tipo de código é um código, que está se tornando padrão entre as linguagens reativas síncronas, sendo conhecido como formato OC. OC é uma abreviação de *Object Code* e trata-se de um formato de arquivo que descreve todas as características de um autômato, como os seus sinais de entrada e saída, os sinais internos e também as ações a serem executadas. Esse formato foi criado para manter um vínculo entre as linguagens síncronas, pois tanto Esterel, quanto Lustre, geram arquivos neste formato.

Para facilitar a interligação com outros componentes, o Compilador RS 5.0 foi adaptado no sentido de gerar código para uma linguagem seqüencial. Assim é gerado um programa codificado em C que faz a simulação do autômato. Esse código C é gerado para programas RS distribuídos ou não. No caso de sistemas não-distribuídos, será gerado um programa C, padrão ANSI, que poderá ser compilado e executado em qualquer plataforma, desde que a mesma possua um compilador C-ANSI. Para sistemas distribuídos, será gerado um código mais complexo. O compilador criará os arquivos RS_Main e RS_IO, além dos arquivos referentes a cada um dos autômatos especificados no código fonte RS.

Pode-se dizer, em suma, que a distribuição da Linguagem RS, além de ampliar a capacidade de RS, permitindo a distribuição do controle, faz com que a programação e a distribuição desses autômatos seja facilitada. Estando todo o código em apenas um arquivo fonte, como ocorre na linguagem SR, aumenta-se a visão do programador quanto à estrutura de todo o sistema. Além disso, a utilização da linguagem C permite que a programação da camada de manipulação de dados seja facilitada, pois permite utilizar diversas bibliotecas já existentes.

As experiências realizadas com a linguagem até aqui permitem esperar que RS venha a se constituir, de fato, numa ferramenta útil para o projeto e a construção de sistemas reativos.

7.1 Trabalhos Futuros

No futuro, pretende-se melhorar a linguagem nos pontos em que a mesma se mostrar fraca. A utilização da linguagem no desenvolvimento de aplicações práticas de maior porte, poderá mostrar novas necessidades ou aperfeiçoamentos, os quais serão introduzidos em versões subsequentes. A criação de um ambiente de execução mais apropriado para a simulação e teste de programas também é proposta como trabalho futuro, pois o Compilador RS 5.0 não possui essa propriedade que era incorporada à versão 4.0.

É natural que haja aperfeiçoamentos a serem introduzidos no ambiente RSD. Um deles diz respeito à segurança, pois a queda do RS_Main faz com que todos os autômatos fiquem à espera de uma mensagem que nunca chegará. A verificação do funcionamento do mestre deve ser implementada.

Certamente devem ser feitas melhorias nas próximas implementações do novo núcleo MDX-RS como o Gerenciador de Execução. Por enquanto, cada autômato RSD deve ser disparado manualmente em cada uma das máquinas do ambiente. Um trabalho a ser desenvolvido deveria tratar da criação de um gerenciador que se responsabilizasse por fazer com que o núcleo e os autômatos fossem executados em máquinas remotas com apenas um comando na máquina onde o cliente está logado.

Outro possível trabalho seria em relação a testes de desempenho. Não foi realizada uma bateria de testes para a comprovação de resultados que dariam garantia a este novo modelo. A única avaliação que foi feita refere-se a apenas uma verificação visual do programa que simula um *mouse* distribuído executando sobre o novo e sobre o antigo núcleo do MDX. O novo núcleo obteve resultados melhores do que o antigo, e semelhantes aos resultados da versão RS 4.0. Mas com o aumento do tamanho e complexidade dos programas RSD, este novo núcleo deve confirmar a sua superioridade, em matéria de desempenho.

A experimentação do núcleo de comunicação MDX-RS, em aplicações distribuídas reais de autômatos RS, provavelmente indicará outras melhorias a serem introduzidas tanto no modelo proposto como na sua implementação.

Bibliografia

[AND 93] ANDREWS, Gredory R.; OLSSON, Ronald A. **The SR Programming Language**: Concurrency in Practice. [S.l.]: The Benjamin / Cummings Publishing Company, 1993.

[AHO 86] AHO, Alfred; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores, princípios, técnicas e ferramentas.** [S.l.]: Addison-Wesley, 1986. 796p.

[BER 89] BERRY, G. **Real Time Programming**: Special Purpose or General Purpose Languages, Sophia-Antipolis: INRIA, 1989. (Research Report 1065).

[BER 92] BERRY, G. GONTHIER G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. **Science of Computer Programming**, [S.l.], v.19, n.2, p.87-152, 1992.

[BIR 84] BIRREL, A.; NELSON, B. Implementing Remote Procedure Call. **ACM Trans. Computer Systems**, New York: v.2, n.1, p.39-59, Feb. 1984.

[BOU 91] BOUSSINOT, F.; SIMODE, R. de. **The Esterel Language**. Sophia-Antipolis: INRIA, 1991. (Research Report, 1487).

[CAM 96] CAMPANI, Carlos A. P. **Linguagens Formais**. Pelotas : Universidade Federal de Pelotas, 1996. 35 p. Polígrafo da disciplina de Linguagens Formais.

[CAM 97] CAMPANI, Carlos A. P. **Compiladores**. Pelotas : Universidade Federal de Pelotas, 1997. 55 p. Polígrafo das disciplinas de Compiladores I e Compiladores II.

[CCK 99] COPETTI, Alessandro; COSTA, Celso M. da; KANNAT, Salah Eddine. **Arquitetura e Implementação do Sistema MDX com a tecnologia de Comunicação ATM.** Porto Alegre: PPGCC da PUCRS, 1999.

[CLA 83] CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. **Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications**: A Practical Approach. [S.l.]: Carnegie-Mellon University, 1983.

[COP 2000] COPETTI, Alessandro; COSTA, Celso da. **MDXv1 – Implementação e Avaliação em Redes ATM**. Porto Alegre: PPGCC da PUCRS, 2000. Dissertação de Mestrado.

[COS 93] COSTA, Celso M. da; FAVRE, Michel; BRIAT, Jacques. **Implementação de um Mecanismo de RPC em uma Máquina Paralela sem Memória Comum**. Paris: Labomouseire de Génie Informatique, PLoSys, 1993.

[COS 96] COSTA, Celso Maciel da. Microkernel Paralelo: Concepção e Implementação em uma Máquina Paralela sem Memória Comum. In: JAIOO, 25., 1996, Buenos Aires. **Anais...** Buenos Aires: [s.n.], 1996.

[COU 98] COURONNE, P. The LUSTRE-ESTEREL Portable Format – In: Ecole Nationale Superieure Des Mines. Paris. **Proceedings...** Paris: INRIA, Sept. 1998.

[FEL 95] FÉLIX, Cláudio Antônio. **Sistemas de Memória Compartilhada**: Um Estudo Comparativo. Porto Alegre: CPGCC da UFRGS, 1995.

[GIO 97] GIORGI, Ulisses Ponticelli. **Estudo para a definição de uma Interface de comunicação para autômatos da Linguagem RS**. Porto Alegre: CPGCC da UFRGS, 1997. (TI-600).

[GIO 98] GIORGI, Ulisses Ponticelli. **A Distribuição da Linguagem RS.** Porto Alegre: CPGCC da UFRGS, 1998. Dissertação de Mestrado.

[HAL 85] HAREL, D.; PNUELI, A. On the Development of Reactive Systems. In: APT, K. R. (Ed.) **Logics and Models of Concurrent Systems**. Berlin: Springer-Verlag, 1985. (NATO Series, v.F13).

[HAL 91] HALBWACHS, N. et al. The synchronous data flow programming language Lustre. [S.l.]: **Proceedings of the IEEE**, New York, v.79, n.9, p.1305-1320, Sept. 1991.

[HAL 93] HALWACKS, Nicolas. **Synchronous Programming of Reactive Systems**. Dordrecht: Klumer Academic, 1993.

[HEN 90] HENNESSY, M. **The Semantics of Programming Languages:** An Elementary Introduction Using Structural Operational Semantics. [S.l.]: Wiley Press, 1990.

[HOP 79] HOPCROFT, J. E.; HULLMAN, J. D. **Introduction to Automata Theory, Languages and Computation**. Reading, Massachusetts: Addison-Wesley, 1979.

[JOH 74] JOHNSON, Stephen C. **Yacc – Yet Another Compiler-Compiler**, Murray Hill: Bell Laboratories, 1974.

[LE 91] LE GUERNIC, P. et al. **Programming real time applications with SIGNAL**. Paris: INRIA, 1991. (Research Report 1446).

[LESK 74] LESK, E. M.; SHIMIDT, E. **Lex – a lexical analyser generator**. Murray Hill: Bell Laboratories, 1974.

[LI 88] LI, Kay. IVY: A Shared Virtual Memory System for Parallel Computing. In INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1988. **Proceedings...** [S.l.:s.n.], 1988. P.94-101.

[LI 97] LI, Victor O. K.; LIAO, Vanjiun**.** Distributed Multimedia Systems. In: **Proceedings of the IEEE**, New York, v.85, n.7, 1997.

[LIB 2000] LIBRELOTTO. Giovani; TOSCANI, Simão; MONTEIRO, Luís F. Distribution of the RS Language over the MDX Environment. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 4., 2000, Recife. **Anais...** Recife: SBC/UFPE, 2000. p.120-133.

[MAT 99] MATTOS, Júlio. **Proposta de geração de código VHDL a partir da Linguagem RS.** Porto Alegre: CPGCC da UFRGS, 1999. (TI-812).

[MEN 97] MENEZES, Paulo F. B. **Linguagens formais e autômatos**. Porto Alegre : Sagra Luzzato, 1997. 168 p.

[PAR 97] PARK, Sung-Yong. HARIRI, Salim. **A high performance message passing system for network of workstations.** [S.l.:s.n.], 1997.

[PET 77] PETERSON, James. Petri Nets. **Computing Surveys,** [S.l.], v.9, n.3, p.223-252, Sept. 1977.

[PET 85] PETERSON, James. SILBERSCHATZ, Abraham. **Operating Systems Concepts**. New York: Addison-Wesley, 1985.

[PRE 96] PREUSS, Evandro. **Núcleo de Comunicação Multi-Thread**: Definição e Implementação. Porto Alegre: PPGCC da PUCRS, 1996. TI-I.

[PRE 96a] PREUSS, Evandro. **Núcleo de Comunicação Confiável para um Ambiente de Execução Distribuído**. Porto Alegre: Mestrado em Informática. PUC-RS, 1996. TI-II.

[PRE 98] PREUSS, Evandro. **MDX**: Um Ambiente de Programação Paralela baseado em Memória Virtual Distribuída. Porto Alegre: PPGCC da PUCRS, 1998. Dissertação de Mestrado.

[PRI 2000] PRICE, Ana Maria; TOSCANI, Simão S. **Compiladores - Implementação de Linguagem de Programação**. Porto Alegre: Sagra Luzzato, Universidade Federal do Rio Grande do Sul, 2000.

[QUE 82] QUEILE, J-P.; SIFAKIS, J. Specification and Verification of Concurrent Systems in CESAR. In: INTERNATIONAL SYMPOSIUM ON PROGRAMMING, 1982, New York. **Proceedings...** New York: Springer-Verlag, 1982. (Lecture Notes in Computer Science, 137).

[SIL 88] SILBERSCHATZ, Abraham; PETERSON, James. **Operating Systems Concepts**. New York: Addison-Wesley, 1988.

[SCH 90] SCHILD, Herbert. **Turbo C Avançado – Guia do Usuário**. São Paulo: McGrawHill, 1990.

[STA 95] STALLINGS, William. **Operating Systems**. Englewood Clifs, New Jersey: Prentice Hall, 1995.

[TAN 95] TANNENBAUM, Andrew S. **Distributed Operating Systems**. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[TOS 93] TOSCANI, Simão S. **RS**: Uma Linguagem para a Programação de Núcleos Reactivos. Lisboa: Universidade Nova de Lisboa, 1993. Tese de Doutorado.

[TOS 96] TOSCANI, Simão S. **Introdução aos Sistemas Reativos**. Porto Alegre: CPGCC da UFRGS, 1996.

[TRE 85] TREMBLAY, Jean-Paul, SORENSON, Paul G. **The Theory and Pratice Compiler Writing.** [S.l.]: McGraw-Hill, 1985.

[VER 86] VERGAMINI D.; **Verification by Means of Observational Equivalence on Automata**. Paris: INRIA, 1986. (Research Report 501).