UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Um Ambiente de Execução para a Linguagem RS

por

FELIPE ZSCHORNACK

Monografia submetida à avaliação
como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

Prof. Gil Carlos Rodrigues Medeiros, MSc.
Orientador

Prof. Júlio Carlos Balzano de Mattos, MSc.
Co-Orientador

Pelotas, abril de 2003.

FELIPE ZSCHORNACK

UM AMBIENTE DE EXECUÇÃO PARA A LINGUAGEM RS

Monografia de Conclusão de Curso
Universidade Federal de Pelotas
Instituto de Física e Matemática
Curso de Ciência da Computação

Orientador: Gil Carlos Rodrigues Medeiros, MSc
Co-Orientador: Júlio Carlos Balzano de Mattos, MSc

Pelotas

2003

UM AMBIENTE DE EXECUÇÃO PARA A LINGUAGEM RS

Monografia defendida e aprovada em 14 de abril de 2003
pela banca examinadora composta pelos professores:

____
Prof. Gil Carlos Rodrigues Medeiros, MSc. (UFPel)

____
Prof. Júlio Carlos Balzano de Mattos, MSc. (ULBRA)

____
Prof. José Luiz Almada Güntzel, Dr. (UFPel)

____
Prof. Simão Sirineo Toscani, Dr. (PUC-RS)

DEDICATÓRIA

Ao Pai Celeste, em primeiro lugar, pois a ele devo tudo o que sou e o que tenho, e também porque, sem Ele, nada valeria a pena...

A meus pais, Lidio e Elmira, não tenho palavras suficientes para expressar minha gratidão pelos esforços que fizeram e pelas dificuldades pelas quais passaram para que eu pudesse estudar e 'ser alguém na vida'. Apesar da grande distância e da imensa saudade, vocês sempre estavam comigo em meus pensamentos. Tudo o que vocês me ensinaram está em minha mente e, sobretudo no meu coração. Amo muito vocês!

Aos manos, Fábio, Tiago e Talita, e à cunhada Denise, por tudo que vivemos nestes últimos anos, por terem me suportado e ajudado em todos os momentos. Vocês todos moram em meu coração!

AGRADECIMENTOS

Ao Gil, meu orientador e amigo de todas as horas, pelos conselhos e ensinamentos que me passou, não somente sobre computação, mas também, e principalmente, sobre a vida. Teu exemplo de caráter e de profissionalismo me será importantíssimo!

Ao Professor Simão Toscani, sempre atencioso para sanar minhas dúvidas quando estas surgiam. Sem sua ajuda certamente este trabalho não poderia ser concretizado. Obrigado!

Ao pessoal da Congregação 'Da Redenção', especialmente aos meus grandes amigos 'Chora', 'Samuca', 'Chorinha', 'Enrolão', 'Luisinho', 'Cabeçudo', Dona Aura e pessoal do 'Navega', galera do Solideo e do coral, enfim, todas aquelas pessoas que me são especiais, por me acolherem como a um filho. Vocês fazem parte da minha grande Família!

Ao 'seu' Arthur e a 'dona' Erna, meus queridos avós, pelo apoio e dedicação em todos os momentos, principalmente nos primeiros tempos de solidão na Cohabpel... Muito Obrigado!

Ao Rochedo, pelo inestimável auxílio na parte de programação utilizando Delphi. Muito Obrigado!

Ao Jaris, pela eterna paciência com este colega 'menos experiente' no mundo da computação. Valeu pela força, e boa sorte no teu mestrado!

Aos demais colegas da faculdade, pelo companheirismo e amizade de todas as horas e, também, pelos jogos 'memoráveis' e pelos 'churras' que fizemos.

SUMÁRIO

SUMÁRIO 6
LISTA DE ABREVIATURAS 8
LISTA DE FIGURAS 9
LISTA DE TABELAS 11
RESUMO 12
1 Introdução 13
2 Sistemas Reativos 16
2.1 Considerações Preliminares 16
2.2 Organização de um programa reativo 17
2.3 Concorrência x Determinismo 18
2.4 Hipótese de Sincronismo 19
2.5 Ferramentas para Programação 19
3 A Linguagem RS 21
3.1 Introdução à Linguagem RS 21
3.2 Construções da linguagem 21
3.2.1 Sinais 22
3.2.2 Variáveis 23
3.2.3 Operadores e comandos 23
3.3 Sintaxe 25
3.4 Semântica 27
3.4.1 O algoritmo de execução 28
3.5 Exemplo de um programa RS 29
4 Ambiente de Execução RS 31
4.1 Compilador 32
4.2 Estrutura do Código 33
4.3 Simulador 37
5 Considerações sobre interfaces de usuário 40
5.1 Definição de Interface 40
5.2 Processo de Interação 41
6 O Novo Ambiente de Execução 43

6.1 Aspectos Gerais de Projeto 43
6.2 Atualização do Interpretador RS 44
6.3 O Módulo de Carga / Execução 46
6.4 A Interface Visual 47
7 Implementação do Módulo Executor 49
7.1 A Memória 49
7.2 O Carregador 52
7.2.1 Reconhecimento e Armazenamento do Autômato 54
7.2.2 Reconhecimento e Armazenamento das Regras de Transição 56
7.3 O Executor 57
8 Implementação de uma Interface Genérica 59
8.1 A Interface Visual Genérica 59
8.2 O Modelo de Comunicação Executor – Interface 65
9 Conclusões 67
ANEXO 1 – Manual de Utilização 69
Bibliografia 75

LISTA DE ABREVIATURAS

RS	Reativa Síncrona
WIMP	Windows, Icons, Menus and Pointers

LISTA DE FIGURAS

Figura 1 - Estrutura Geral de um Módulo RS 25
Figura 2 - Sintaxe de Regras de Reação 26
Figura 3 - Sintaxe de Regra Ação Condicional (Case) 27
Figura 4 - Algoritmo de Execução RS 28
Figura 5 - Exemplo de um Programa RS 29
Figura 6 - Esquema original do Ambiente de Execução RS 31
Figura 7 - Ambiente após a compilação 33
Figura 8 - Sintaxe do Autômato RS 35
Figura 9 - Sintaxe de Regras Condicionais 36
Figura 10 - Código Gerado para Regras de Transição 36
Figura 11 - Exemplo de Código Objeto RS 37
Figura 12 - Utilização do comando 'sar' 39
Figura 13 - Esquema de projeto do novo ambiente de execução 43
Figura 14 - Diagrama do módulo executor 49
Figura 15 - Estrutura da memória 50
Figura 16 - Estrutura da Tabela de Variáveis 50
Figura 17 - Estrutura da Tabela de Sinais 51
Figura 18 - Esquema de Carga para Memória 53
Figura 19 - Esquema da Tabela de Transições do Autômato 54
Figura 20 - Esquema da Tabela de Regras 56
Figura 21 - Janela para inserção de valor em campo de sinal 60
Figura 22 - Janela Principal após seleção de programa 62
Figura 23 - Indicação de erro de carga 62
Figura 24 - Janela de Visualização de Erros 62
Figura 25 - Menu para visualização da Lista de Erros 63
Figura 26 - Visualização das Tabelas 63
Figura 27 - Janela de Execução 64
Figura 28 - Menu de contexto para sinais 65
Figura 29 - Janela de abertura de arquivo 70
Figura 30 - Janela de Configuração 70

Figura 31 - Botões disponíveis 71
Figura 32 - Menu Arquivo 71
Figura 33 - Modos de Execução 72
Figura 34 – Configuração de Escala 72
Figura 35 – Botões de execução 73
Figura 36 - Botões para entrada de sinal no modo Interativo 74

LISTA DE TABELAS

Tabela 1 - Operadores válidos da linguagem RS-0 24
Tabela 2 - Comandos da Linguagem RS-0 24
Tabela 3 - Comandos avançados da linguagem RS 25

RESUMO

Uma importante classe de problemas computacionais é tratada por sistemas caracterizados como reativos, cujo controle pode ser descrito através de linguagens especialmente definidas para esta finalidade. A linguagem RS é uma ferramenta adequada para descrever tais sistemas. Ela é composta de duas partes: um compilador para geração de autômatos e um simulador de execução com facilidades para depuração de programas.

Este trabalho apresenta um novo executor para o código objeto (autômato finito + regras de transição) gerado pelo Compilador RS, mais especificamente para a sua versão RS-0. Tal executor independe da linguagem Prolog, utilizada na execução de programas RS na versão original. Junto a este novo simulador, foi agregada uma interface mais amigável que a anteriormente utilizada, a qual facilita a utilização do executor para a simulação de programas RS.

Na elaboração do executor, foi levado em consideração o formato do código objeto gerado pelo compilador da linguagem RS. Tal código fornece ao executor todas as informações necessárias para a montagem da interface com o usuário e para a execução do código propriamente dita.

Este trabalho também descreve o protocolo de comunicação utilizado entre o executor e a interface genérica. Este protocolo pode ser tomado como base para o desenvolvimento futuro de interfaces mais específicas que tenham como base o presente executor.

PALAVRAS-CHAVE: Sistemas Reativos, Linguagem RS, Compiladores, Ambiente de Execução, Interface.

1 Introdução

Uma importante classe de problemas computacionais é tratada por sistemas caracterizados como **reativos**. De acordo com Halbwachs [HAL 93] e Toscani [TOS 93], são considerados Sistemas Reativos os sistemas computacionais que reagem continuamente a estímulos oriundos do ambiente em que estão inseridos. Segundo Berry (apud [TOS 93]), tais sistemas têm sua velocidade determinada pelo ambiente e não pelo próprio sistema. Exemplos de sistemas reativos estão em toda a parte: controladores de processos industriais, videogames, interfaces de usuário, relógios digitais de múltiplas funções, etc.

No contexto dos sistemas reativos, é possível enxergá-los como possuindo três camadas distintas: uma camada de **núcleo reativo**, responsável por realizar as reações em função dos estímulos oriundos do ambiente externo; uma camada de **interface**, que faz o intercâmbio de informações entre o ambiente externo e o núcleo; e uma outra camada de **manipulação de dados,** que realiza computações triviais requeridas pelo núcleo reativo, normalmente implementadas por procedimentos externos escritos em outra linguagem. Dentre todas essas, a camada de núcleo reativo é considerada como sendo a parte central e mais difícil de um programa reativo.

Várias ferramentas para desenvolvimento de tais sistemas foram desenvolvidas, tais como Esterel e Lustre [HAL 93]. Uma delas, a Linguagem RS, implementada por Toscani, foi desenvolvida especificamente para a programação de **núcleos reativos**.

Na construção da Linguagem RS, que é composta por um compilador e um simulador com facilidades para depuração de programas, foi utilizada a linguagem de programação Prolog. O uso de tal linguagem facilitou em muito a elaboração de RS, tendo em vista que a sintaxe de RS foi fortemente influenciada pela sintaxe de “termos” do Prolog.

Porém, o fato de ser desenvolvida através do uso do Prolog trouxe também influências negativas para a linguagem RS: além da elaboração do compilador, foi necessária a criação de um **interpretador** (ou **executor**) para seu código compilado, visto que o Prolog não possuía facilidades para geração de programas executáveis.

Assim, o usuário que desejar utilizar um programa RS deve compilar o código e, então, fazer uso do interpretador para a execução do programa. Todo este processo deve ser feito com o ambiente Prolog ativo, pois ele é necessário para a execução do compilador e do executor RS.

Por outro lado, um software que possua uma interface amigável permite que o usuário se sinta bem ao utilizá-lo, descobrindo as potencialidades do mesmo de maneira muito mais intuitiva. Ou seja, a própria natureza da interface deve, por si só, levar o usuário a ter uma idéia do que ele poderá fazer. O interpretador da linguagem RS, em seu formato original, não dispõe desta facilidade, exigindo do usuário o conhecimento de seus comandos para sua utilização.

Além disso, seria interessante que a simulação de um programa RS pudesse contemplar a entrada de todos os sinais do programa e dos instantes em que cada um dos sinais devesse ocorrer. Isto possibilitaria uma menor interação do usuário com o sistema, tornando o processo de execução do programa mais realístico, conforme destacado por Toscani [TOS 93].

Assim, o presente trabalho apresenta os seguintes objetivos:

- Complementar conhecimentos, sobretudo nas áreas de Sistemas de Computação e de Linguagens de Programação, através do estudo das características dos sistemas reativos e da linguagem reativa RS, de forma a prover informações que possibilitem o futuro prosseguimento de pesquisas nessa área;
- Modelar e implementar um novo executor para o código objeto gerado pelo compilador da linguagem RS, o qual deve ser independente do ambiente Prolog;
- Modelar e implementar uma interface visual para o novo executor, que seja mais agradável e facilite a utilização do sistema pelo usuário. Tal interface deverá prover elementos que permitam ao usuário simular diretamente um programa RS, através da inserção dos sinais de entrada desejados e dos instantes de tempo para tais sinais. Também deverá permitir ao usuário executar um programa RS passo a passo, a fim de poder verificar seu comportamento de forma mais apurada;

- Elaborar um protocolo de comunicação entre a interface visual e o executor, que descreva as informações que o executor deverá receber para seu correto funcionamento e os dados repassados por este para a interface. Este protocolo deverá possibilitar o desenvolvimento de novas interfaces mais específicas baseadas no novo executor.

Esta monografia está organizada da seguinte maneira. No Capítulo 2, é feita uma breve introdução aos sistemas reativos, abordando suas principais características.

A apresentação da Linguagem RS, com especial destaque para sua sintaxe e semântica, é feita no Capítulo 3. O Capítulo 4, por sua vez, descreve o ambiente de execução atualmente utilizado e a estrutura do código objeto RS. O Capítulo 5 é dedicado a um breve estudo sobre interfaces, mostrando a necessidade de criação de um novo ambiente que seja mais amigável em termos de interface do que o ambiente original.

A modelagem do novo ambiente de execução, tanto em termos da interface visual quanto do executor propriamente dito, é abordada no Capítulo 6. Neste mesmo capítulo, é feita uma breve referência sobre os ajustes efetuados na versão 4.1 da linguagem RS, que foi utilizada no presente trabalho para a realização de testes.

A implementação da máquina virtual RS, composta pelos módulos de carga, execução e memória, é o assunto tratado no Capítulo 7. O capítulo seguinte descreve a implementação da interface genérica utilizada no ambiente de execução implementado. Este capítulo também apresenta o modelo de comunicação entre a interface genérica e o módulo executor RS.

O Capítulo 9 apresenta as conclusões obtidas a partir deste trabalho, apontando possibilidades de trabalhos futuros baseados neste.

Por fim, o Anexo 1 apresenta um manual para a utilização do novo ambiente implementado, enfatizando o processo de simulação de um programa RS.

2 Sistemas Reativos

Este capítulo apresenta, de forma sucinta, informações relevantes a respeito de sistemas reativos. A parte inicial destaca algumas diferenças entre os diversos tipos de sistemas existentes. Também é discutida a organização de um programa reativo, aspectos sobre concorrência, determinismo e sincronismo, bem como as principais ferramentas de programação existentes.

2.1 Considerações Preliminares

Ao longo dos anos, os sistemas computacionais foram sendo divididos em várias classes, de acordo com o comportamento observado em cada um deles. Assim, surgiram diversas dicotomias que são bastante utilizadas para classificar os diversos sistemas existentes. Essas dicotomias distinguem, de acordo com as características apresentadas, Sistemas Seqüenciais de Sistemas Concorrentes, Sistemas Determinísticos de Sistemas Não-Determinísticos, etc.

Com o passar do tempo e com o crescimento da complexidade no desenvolvimento desses sistemas, observou-se que era possível estabelecer uma nova dicotomia. Assim, David Harel e Amir Pnueli [HAR 85] propuseram uma nova divisão que pudesse contemplar as diferenças entre os sistemas relativamente fáceis de desenvolver e os de difícil tratamento ou problemáticos. Desta forma, a nova dicotomia proposta distingue os Sistemas Transformacionais dos Sistemas Reativos.

Segundo Toscani [TOS 93], os *sistemas transformacionais* são definidos como aqueles sistemas que obtém suas respostas a partir de um conjunto de dados de entrada. De forma semelhante, pode-se dizer que tais sistemas comportam-se como sendo funções dos seus dados de entrada, realizando computações sobre os mesmos e devolvendo respostas a partir da transformação destes [MAT 2000].

Da mesma forma, podem ser considerados transformacionais os sistemas que, no decorrer do processamento dos dados de entrada, possam requisitar novos dados, bem como aqueles que, de alguma forma, possam produzir parte de suas saídas no

decorrer de suas execuções. Exemplos amplamente conhecidos deste tipo de sistema são os compiladores e os programas que resolvem problemas numéricos.

Os *sistemas reativos*, por sua vez, são designados como aqueles "que, continuamente, devem responder a estímulos provenientes de um ambiente externo, numa ordem desconhecida", conforme Toscani [TOS 93].

Estes sistemas possuem características bem peculiares. Eles se relacionam de forma dinâmica com o ambiente em que estão inseridos, interagindo de maneira muito forte com este ambiente. Além disso, a velocidade desta interação é determinada pelo ambiente e não pelo sistema, o que diferencia os sistemas reativos dos sistemas interativos, conforme Berry (apud [TOS 93]). Os *sistemas interativos* continuamente interagem com seu ambiente, de maneira semelhante aos sistemas reativos. Contudo, o ritmo desta interação é ditado pelo próprio sistema e não pelo ambiente. Os sistemas "time-sharing" ilustram este tipo de sistema.

Existem, além dos sistemas reativos convencionais, sistemas reativos especiais denominados *sistemas de tempo real*. Tais sistemas devem responder dentro de limites estritos de tempo e estão, normalmente, envolvidos em aplicações críticas, como no controle de usinas nucleares, nos quais os processos externos dependem dos tempos de resposta do computador. Em geral, sistemas de tempo real são reativos, porém o contrário nem sempre é verdadeiro, uma vez que é comum existirem sistemas reativos para os quais os tempos de resposta não são cruciais. Um exemplo disso são os "drivers" de sistemas operacionais [ARN 98].

Pode–se visualizar o comportamento de um sistema reativo como sendo uma seqüência de três passos executados de maneira cíclica:

- espera por um estímulo vindo do ambiente externo;
- determinação das respostas, de acordo com o estímulo recebido;
- emissão de sinais de saída.

Os sinais de saída emitidos retornam ao ambiente externo, influenciando-o.

Como exemplos de sistemas reativos, são citados os controladores de processos industriais, relógios digitais de múltiplas funções, videogames, máquinas de venda automática, dentre outros.

2.2 Organização de um programa reativo

De forma geral, um programa reativo é organizado em três camadas, a saber:

- *Interface* – encarregada de receber estímulos e encaminhar saídas ao ambiente, transformando eventos físicos externos em sinais lógicos internos, entendíveis pelo núcleo reativo, e vice-versa.
- *Núcleo Reativo* – contém a lógica do sistema. Aqui, as entradas e saídas lógicas são manipuladas e as reações do sistema são realizadas. Isto significa dizer que, para cada sinal de entrada recebido através da camada de interface, o núcleo realiza computações e pode gerar saídas para o ambiente externo, as quais serão repassadas para este através da mesma camada de interface.
- *Manipulação de Dados* – executa computações triviais requeridas pelo núcleo reativo. Normalmente são implementadas por procedimentos externos escritos em outra linguagem.

A organização através de camadas, descrita acima, permite que os núcleos reativos, considerados como sendo a parte central e mais difícil de um programa reativo, possam ser trabalhados, ou seja, estudados, modelados e implementados, de forma completamente separada do ambiente com o qual interagem. Esta abordagem permite desconsiderar totalmente os detalhes de comunicação com o ambiente.

2.3 Concorrência x Determinismo

Determinismo é uma característica importante dos sistemas reativos. Um programa reativo determinístico sempre produz seqüências de saídas idênticas quando estimulado com idênticas seqüências de entrada. Além disso, sistemas determinísticos são mais simples em termos de especificação, depuração e análise em relação a sistemas não-determinísticos [MAT 2000].

Sistemas puramente seqüenciais são, naturalmente, determinísticos. Porém, determinismo não implica seqüencialidade. A maioria dos sistemas reativos pode mesmo ser decomposta em subsistemas concorrentes determinísticos que cooperam de maneira determinística. Por exemplo, um relógio de pulso digital típico contém um relógio propriamente dito, um cronômetro e um alarme, todos naturalmente cooperando de maneira determinística.

Assim, concorrência determinística é a chave para o desenvolvimento de programas reativos e somente é suportada por linguagens síncronas [TOS 93].

2.4 Hipótese de Sincronismo

Segundo Caspi [CAS 94], as linguagens síncronas foram introduzidas nos anos 80 para tornar a programação de sistemas reativos mais fácil. Tais linguagens baseiam-se na *hipótese de sincronismo*, a qual presume que todas as computações ocorrem em passos atômicos discretos durante os quais o tempo é ignorado. Dessa forma, assume-se que o código do programa executa em tempo zero. O tempo somente avança quando nenhum código está disponível para execução. Durante um passo simples, considera-se que as saídas ocorrem imediatamente após as entradas, isto é, a saída é síncrona com a entrada. O código executado em cada passo é chamado de *reação*.

As linguagens síncronas são, portanto, linguagens ideais, cujos comandos são executados em tempo zero. Também é possível garantir a hipótese de sincronismo, considerando os comandos da linguagem como sendo reais e o processador como sendo infinitamente rápido [TOS 93]. Naturalmente, um tal processador infinitamente veloz não existe. Entretanto, o que importa é que o ambiente externo permaneça inalterado (congelado) durante o tempo da reação, ou seja, é suficiente que qualquer entrada seja tratada totalmente antes da chegada de uma próxima entrada.

2.5 Ferramentas para Programação

Conforme Arnold [ARN 98] e Mattos [MAT 2000], existem diversas ferramentas que podem ser utilizadas para a programação de sistemas reativos. Dentre as principais, podem ser destacadas as seguintes:

- *Autômatos Determinísticos*: são máquinas abstratas formadas por um conjunto de estados. A partir do estado atual do autômato e da informação de entrada, existe uma e somente uma possibilidade de transição para outro estado do autômato. O estado atual é função das informações anteriores e o próximo estado é função do estado atual e da informação presentemente fornecida. São utilizados para a programação de núcleos reativos de pequeno porte. Permitem obter excelente desempenho em tempo de execução e são bem conhecidos matematicamente. Entretanto, existem alguns problemas quanto à sua utilização: o projeto e a manutenção de autômatos determinísticos é uma tarefa difícil e muito sujeita a erros. Além disso, os autômatos não suportam concorrência, pois são essencialmente seqüenciais.

- *Linguagens Concorrentes*: oferecem as maiores facilidades para a programação de sistemas reativos. Algumas possuem recursos tão poderosos que permitem definir todas as três camadas de um sistema reativo de forma direta, o que, em geral, não acontece com as outras ferramentas. Isto ocorre porque boa parte das ferramentas para programação de sistemas reativos não é de propósito geral nem tampouco auto-suficiente, sendo as camadas de interface e manipulação de dados especificadas em alguma linguagem hospedeira, conforme Toscani [TOS 93]. As linguagens concorrentes permitem o desenvolvimento hierárquico e modular de programas, mas possuem o inconveniente de serem não-determinísticas.
- *Linguagens Síncronas*: são baseadas na hipótese de sincronismo. Surgiram para permitir uma programação mais fácil e obter uma semântica mais clara para os sistemas reativos [MAT 2000]. As duas opções anteriores obrigam o projetista a escolher entre concorrência e determinismo. As linguagens síncronas, porém, conciliam as duas idéias, permitindo combinar os benefícios da programação concorrente com a eficiência de execução dos autômatos determinísticos. Dentre as principais linguagens síncronas, destacam-se Lustre , Signal e Esterel, conforme, respectivamente, Caspi, Guernic e Berry (apud [HAL 93]), além da linguagem RS, que será abordada nos próximos capítulos. Informações sobre as outras linguagens podem ser obtidas na publicação citada.

3 A Linguagem RS

Este capítulo é dedicado ao estudo da Linguagem RS e de suas características principais. Além de apresentar uma introdução à linguagem, são discutidos aqui aspectos de sua sintaxe e semântica, bem como de suas principais construções. Também é visto o algoritmo de execução de um programa RS.

3.1 Introdução à Linguagem RS

A linguagem RS foi desenvolvida para a programação de núcleos reativos. Tais núcleos devem reagir de maneira imediata a estímulos oriundos do ambiente em que estão inseridos, os quais chegam, através da interface, em uma ordem desconhecida.

Um programa reativo desenvolvido com a utilização de RS é visto como um sistema que: reage ao receber um estímulo do ambiente, modificando-se internamente; emite sinais de saída para o ambiente externo; e espera por um novo sinal externo. Todo esse processo ocorre de maneira instantânea e determinística. Pelo fato de uma reação ocorrer de forma instantânea, ou seja, por não consumir tempo, o programa reativo estará permanentemente esperando por um sinal ou estímulo.

De forma geral, uma regra de reação na linguagem RS é expressa da seguinte maneira:

```
Sinal & Condição → Ação
```

A semântica da regra de reação mostrada acima é a seguinte: ao chegar o *Sinal* e a *Condição* for verdadeira, a *Ação* será executada.

3.2 Construções da linguagem

Conforme definido por Toscani [TOS 93], a linguagem RS possui quatro versões evolutivas disponíveis, denominadas RS-0 (versão básica, utilizada para prototipar o novo executor proposto neste trabalho), RS-1, RS-2 e RS (Versão Plena).

Na versão básica (RS-0), um programa é representado por um conjunto único de regras de reação. Nas versões posteriores, é permitida a utilização de construções de

linguagem mais sofisticadas, tais como módulos, caixas de regras, sensores e variáveis do tipo *record*.

Tendo em vista que a sintaxe para a versão RS-0, apresentada por Toscani [TOS 93], não mais é suportada pelas versões atuais da linguagem (como a versão 4.1, utilizada para os testes), as construções e a sintaxe aqui apresentadas serão direcionadas no sentido de demonstrar as adaptações necessárias para possibilitar a criação de um programa que gere um código compilado RS compatível com a sintaxe de RS-0 original.

Assim, serão apresentadas apenas as construções mais importantes da linguagem RS em sua versão básica. As demais construções podem ser encontradas na bibliografia pertinente ([TOS 93], [MAT 2000], [LIB 2001], [ARN 98] e [GIO 98]).

3.2.1 Sinais

Os sinais, na linguagem RS, podem ser divididos em três grandes grupos:

- *Sinais de Entrada*: correspondem aos eventos produzidos pelo ambiente externo que fornecem ao programa os estímulos necessários para desencadear reações (são os únicos sinais que possuem tal capacidade). Em um programa RS, é obrigatório o uso de, pelo menos, um sinal de entrada, pois não faz sentido um programa reativo que não possa ser estimulado. Para declarar um sinal de entrada no programa, utiliza-se a palavra reservada *input*.
- *Sinais de Saída*: são utilizados pelo programa para indicar ao ambiente externo resultados das reações realizadas. A emissão de um sinal se dá através da utilização do comando *emit* em uma regra de reação. Em sua declaração, utiliza-se a palavra reservada *output*.
- *Sinais Internos*: são sinais utilizados para comunicação e sincronização interna. Para ativar um sinal interno, usa-se o comando *up*. Para usar um sinal interno, utiliza-se a palavra reservada *signal*.

Na versão 4.1, utilizada neste trabalho, a sintaxe dos sinais internos *signal* é substituída por *t_signal* e *p_signal*. A palavra reservada *t_signal* define sinais internos temporários, os quais são utilizados apenas durante o desenvolvimento de uma reação, sendo automaticamente desligados ao seu final. Já *p_signal* define sinais que, quando ligados, permanecem nesse estado até que sejam explicitamente desligados. Assim, podem passar ligados de uma reação para outra.

Assim, para obter o correto funcionamento de um programa reativo neste trabalho, deve-se utilizar *p_signal* ao invés de *signal*, deixando t_signal vazio.

Os sinais podem ou não conter valores. Sinais que não contém valores são ditos puros e os que contém são ditos valorados. Por convenção, os campos de valor de um sinal são identificados por nomes que iniciam por letra maiúscula. Um sinal valorado é especificado pela notação $ns(V_1, ..., V_n)$, onde *ns* é o nome do sinal e V_i $(1 \leq i \leq n)$ são os campos que armazenam valores do sinal.

3.2.2 Variáveis

Na linguagem RS, as variáveis são declaradas a partir da utilização da palavra reservada *var* e podem assumir apenas valores numéricos. Sintaticamente, os nomes das variáveis devem iniciar sempre por letra minúscula. Na versão RS-0 da linguagem, as variáveis somente podem ser simples. Um exemplo de declaração de variáveis é:

```
var: [x, y, z]
```

3.2.3 Operadores e comandos

Segundo Toscani [TOS 93], a linguagem RS sofreu forte influência da sintaxe de termos Prolog. Esta influência é ser percebida através da maneira pela qual os operadores e comandos da linguagem RS foram especificados.

A Tabela 1 e a Tabela 2, respectivamente, ilustram os operadores e os comandos disponíveis em RS-0, descrevendo brevemente a função de cada um, conforme Mattos [MAT 2000]. A Tabela 3 ilustra os comandos que não fazem parte da sintaxe de RS-0 e que estão disponíveis somente nas versões mais elaboradas da linguagem.

Tabela 1 - Operadores válidos da linguagem RS-0

Operador	Descrição	
module	Declaração de módulo	
input	Definição de sinais de entrada	
output	Definição de sinais de saída	
t_signal	Definição de sinais internos temporários	
p_signal	Definição de sinais internos permanentes	
var	Declaração de variáveis	
initially	Comandos a serem executados antes do início do programa	
===>	Vincula ação a uma condição de disparo em uma regra de reação	
--->	Vincula ação a uma condição booleana em regras do tipo 'case'	
:	Vincula uma lista de nomes e/ou comandos a um programa, sinal ou módulo	
#	Indica uma lista de sinais internos	
case	Regra condicional de múltipla escolha	
else	Alternativa default para Case	
~	Não	
&	E	Operadores Lógicos
v	Ou	
=	Igualdade	
=:=	Igualdade	
~=	Diferença	
=\=	Diferença	
>=	Maior ou Igual	Operadores Relacionais
=>	Maior ou Igual	
<=	Menor ou Igual	
=<	Menor ou Igual	
<	Maior	
>	Menor	
+	Soma	
-	Subtração	
*	Multiplicação	
/	Divisão Real	Operadores Aritméticos
//	Divisão Inteira	
mod	Resto da Divisão Inteira	
fix	Conversão Real → Inteiro	
float	Conversão Inteiro → Real	
truncate	Arredondamento de número real	
real_round	Arredondamento de número real (arredondamento inteligente)	
:=	Atribuição	

Tabela 2 - Comandos da Linguagem RS-0

Comando	Descrição
up(*sinal_interno*)	Ativa o sinal interno indicado por *sinal_interno*
emit(*sinal_saída*)	Emite o sinal de saída *sinal_saída* para o ambiente externo

Tabela 3 - Comandos avançados da linguagem RS

Comando	Descrição
activated(*caixa*)	Verifica se a caixa de regras está ou não ativa
exit_to(*caixa*)	Ativa a caixa de regras especificada, desativando aquela na qual o comando é utilizado
deactivate	Desativa a caixa de regras onde o comando é utilizado
signalled(*sinal*)	Verifica se um sinal interno está ou não ligado
get_sig(*sinal*)	Faz a leitura dos valores numéricos de sinais sensores
raise(*evento*)	Sinaliza um evento de exceção durante uma reação
get_rec(*Record*)	Realiza a leitura dos valores de uma variável estruturada
set_rec(*valores*)	Atribui valores aos campos de uma variável estruturada
activate(*rules*)	Habilita a execução de um conjunto de regras agrupadas numa caixa de regras[1]

3.3 Sintaxe

Um programa em RS-0 possui somente um módulo contendo todas as regras de reação do programa. Sua sintaxe tem a forma ilustrada pela Figura 1:

```
module M:
[     input: I,
      output: O,
      t_signal: T,
      p_signal: P,
      var: V,
      initially: C,
      on_exception: E,
      R
].
```

$I = [i_1, ..., i_m]$, $m > 0$, é a lista de sinais de entrada;
$O = [o_1, ..., o_n]$, $n >= 0$, é a lista de sinais de saída;
$T = [t_1, ..., t_z]$, $z >= 0$, é a lista de sinais internos temporários;
$P = [p_1, ..., p_q]$, $q >= 0$, é a lista de sinais internos permanentes;
$V = [v_1, ..., v_x]$, $x >= 0$, é a lista de variáveis;
$C = [c_1, ..., c_s]$, $s > 1$, é a lista de comandos (de inicialização);
$E = [e_1, ..., e_t]$, $t >= 0$, é a lista de regras de exceção;
$R = r_1, ... r_u$, $u > 1$, é a lista de regras de reação.

Figura 1 - Estrutura Geral de um Módulo RS

[1] Se um módulo se reduz a um único conjunto de regras, como no caso de programas RS-0, então o comando "activate", que obrigatoriamente deve aparecer na lista de ações iniciais (initially), deverá ser "activate(rules)", pois "rules" é o nome default da caixa correspondente ao (único) conjunto de regras do módulo.

A ordem das declarações a ser seguida na construção de um programa RS é a mesma indicada na Figura 1. Um programa equivalente, em termos de sintaxe, a RS-0 deve constar de todas as declarações indicadas acima, mesmo que vazias. Nas versões posteriores, isso não é exigido.

De acordo com a sintaxe acima, para que um programa possa ser considerado equivalente à RS-0, nenhum sinal interno temporário ou regra de exceção deve ser declarado. Apesar disso, *t_signal* e *on_exception* devem aparecer na estrutura do programa, ambos constando de uma lista vazia, por se tratar de uma exigência do compilador RS.

A forma geral das regras de reação num programa RS é apresentada na Figura 2a, onde s_e é um sinal de entrada, e s_i ($1 \leq i \leq n$) é um sinal interno e *ações* corresponde a uma lista (possivelmente vazia) de ações a realizar. Se existir um sinal de entrada no lado esquerdo de uma regra de reação, ela é dita *global* ou *externa*, como ilustrado pela Figura 2b. Por outro lado, se existirem somente sinais internos do lado esquerdo da regra, ela é dita *local* ou *interna*, isto é, para estar apta a executar, ela depende somente da habilitação dos sinais internos correspondentes. Esta regra é ilustrada através da Figura 2c.

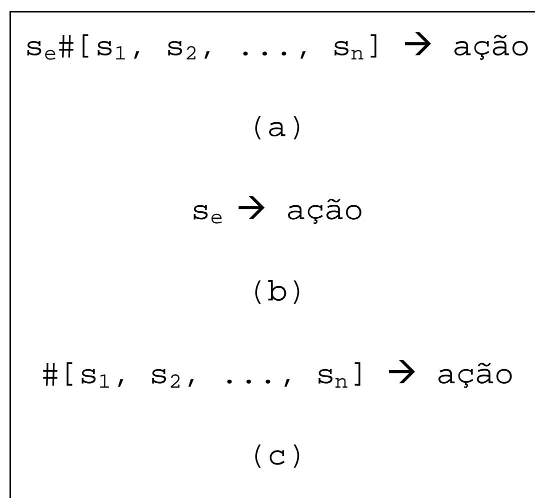

Figura 2 - Sintaxe de Regras de Reação

As ações especificadas nas regras de ação podem ser *incondicionais* ou *condicionais*. As ações incondicionais são executadas sempre que sinais externos e/ou sinais internos estiverem habilitados e não houver nenhuma condição adicional a ser satisfeita. Ações deste tipo são compostas por listas de comandos simples, conforme caracterizado por *C* na Figura 1.

Ações condicionais são aquelas que especificam, pelo menos, duas opções de ação, que são formadas por uma condição e uma lista de comandos simples. A sintaxe deste tipo de ação pode ser vista na Figura 3, onde $\mathbf{b_1, ..., b_n}$ são condições booleanas e $\mathbf{C_1, ..., Cn}$ são listas de comandos simples.

Além disso, é possível utilizar a condição *else* na última opção de uma ação condicional, cujo valor lógico é sempre *true* (ou verdadeiro). Ou seja, os comando de *else* serão executados sempre que ele for especificado e todas condições anteriores não forem satisfeitas. A inexistência de uma condição verdadeira configura um erro de lógica no programa, o qual não é detectado em tempo de compilação.

```
case[b1 → C1,
      ...
      bn → Cn
]
```

Figura 3 - Sintaxe de Regra Ação Condicional (Case)

As condições lógicas especificadas nas ações condicionais, ou *case*, são expressões construídas a partir dos operadores lógicos e relacionais aplicados sobre expressões aritméticas, conforme definidos na Tabela 1. Além disso, é possível a utilização de parêntesis para forçar a precedência desejada sobre determinadas expressões.

3.4 Semântica

Um programa RS é, na verdade, um autômato finito determinístico, pois de acordo com o estado em que se encontra, o programa realiza ações conforme o sinal de entrada que recebe e os sinais internos presentemente habilitados, realizando uma transição para outro estado. Uma regra de reação (Sinal & Condição → Ação) será executada sempre que o conjunto de sinais (tanto internos, quanto externo) especificado do lado esquerdo dessa mesma regra estiver habilitado e a condição (se houver alguma) for verdadeira.

Conforme Mattos [MAT 2000], a forma como os programas RS são representados é conveniente, visto que parece haver concordância geral acerca de que o comportamento de sistemas complexos é naturalmente representado através de estados e eventos.

Para um melhor entendimento da semântica da linguagem RS, é apresentado a seguir o algoritmo geral de execução.

3.4.1 O algoritmo de execução

Na seção 2.1, foi destacado que um programa reativo comporta-se como uma "seqüência de três passos executados de maneira cíclica". Essa seqüência de passos também pode ser vista como um algoritmo de execução, como o que será apresentado nesta seção.

O algoritmo apresentado na Figura 4 está baseado no algoritmo de execução apresentado por Mattos [MAT 2000], excluindo os itens que abordam o tratamento de exceções, o qual não será abordado neste trabalho.

1. Efetuar a ação de inicialização do programa (*initially*).

2. Repetir o seguinte ciclo de execução:

   a. Esperar pelo sinal externo seguinte;

   b. Ao chegar um estímulo externo, repetir as ações *i*, *ii* e *iii* enquanto ainda existirem regras habilitadas para serem executadas:

      i. Identificar quais as regras que podem ser executadas em virtude dos sinais que estão ligados;

      ii. Desligar todos os sinais destas regras identificadas (esses sinais aparecem no lado esquerdo da regra de reação);

      iii. Executar as ações de todas as regras identificadas, em paralelo.

   c. Emitir os sinais para o ambiente externo.

Figura 4 - Algoritmo de Execução RS

A execução de um programa RS é feita de maneira finita e determinística. Isso é garantido por duas condições semânticas definidas na linguagem, que são:

- Jamais uma regra que emita sinal pode ser repetida durante a execução de uma reação;

- Regras paralelas de um passo nunca acessam variáveis de maneira conflitante (uma variável só pode ser acessada por duas ou mais regras simultaneamente se o acesso for para leitura).

3.5 Exemplo de um programa RS

A Figura 5 apresenta um exemplo de programa RS, de acordo com a sintaxe utilizada na versão básica (RS-0):

```
module rst:
[
      input: [r, s, t],
      output: [ok],
      t_signal: [],
      p_signal: [p1, p2],
      var: [],
      initially: [activate(rules)],
      on_exception: [],

r ===> [up(p1)],          /* Regra A */
s#[p1]===> [up(p2)],      /* Regra B */
t#[p1]===> [],            /* Regra C */
s#[p2]===> [],            /* Regra D */
r#[p2]===> [],            /* Regra E */
t#[p2]===> [emit(ok)]     /* Regra F */
].
```

Figura 5 - Exemplo de um Programa RS

Neste exemplo, o programa reconhece seqüências de sinais '*rst*', invariavelmente nesta ordem. Assim sendo, quando o programa receber a seqüência correta de sinais, ou seja, primeiro o sinal *r*, depois o sinal *s* e, por último, o sinal *t*, ele emitirá o sinal *ok,* reconhecendo a seqüência.

O programa da Figura 5 possui seis regras de reação, as quais têm o seguinte significado:

- Regra A – Ao receber o sinal "*r*" oriundo do ambiente externo, o sinal interno "*p1*" é ligado;
- Regra B – Se o programa receber o sinal externo "*s*" e o sinal interno "*p1*" estiver ligado, o sinal "*p2*" é ligado ("*p1*" é desligado quando da execução da regra);

- Regra C – Ao receber o sinal externo "*t*", se "*p1*" estiver ligado, desliga-o e não realiza ação alguma;
- Regra D – Ao receber "*s*", se o sinal interno "*p2*" estiver ligado, o mesmo é desligado sem que nenhuma ação seja realizada;
- Regra E – Quando o programa receber o sinal externo "*r*" e o sinal "*p2*" estiver ligado, o mesmo é desligado e nada ocorre;
- Regra F – Se o programa receber o sinal "*t*" e o sinal interno "*p2*" estiver ligado, este é desligado e o programa emitirá o sinal "*ok*" para o ambiente externo.

4 Ambiente de Execução RS

O ambiente de execução da linguagem RS é composto por dois blocos básicos: um compilador para geração de autômatos e um simulador de execução com facilidades para depuração de programas. Eles foram desenvolvidos através da linguagem Prolog e fazem uso de sua interface de comunicação para interagir com o usuário.

O processo de execução de um programa RS consiste, basicamente, dos seguintes passos principais:

- ativação do ambiente de programação Prolog;
- carga dos módulos correspondentes ao interpretador RS (compilador e executor);
- compilação do programa desejado;
- ativação do simulador para inicio da execução do programa compilado.

Atingido este estágio, é possível iniciar a simulação do programa, visualizar o código objeto gerado, depurar o código ou até mesmo abortar o processo de simulação. Além disso, a partir de uma pequena modificação efetuada no código do interpretador RS realizada neste trabalho, tornou-se possível salvar o código objeto gerado para algum arquivo. A Figura 6 mostra a estrutura original do ambiente de execução RS.

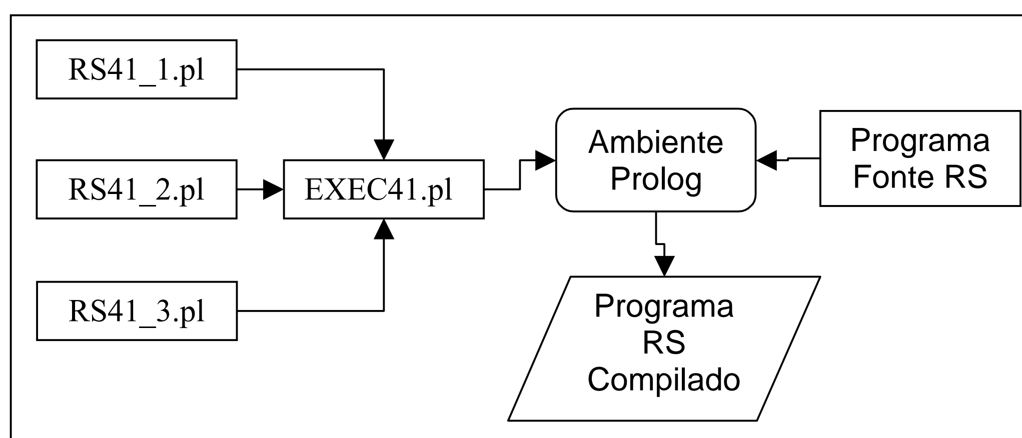

Figura 6 - Esquema original do Ambiente de Execução RS

As próximas seções descreverão, em maiores detalhes, o funcionamento do compilador RS, o formato do código objeto gerado pelo mesmo (que será utilizado

como entrada para o novo ambiente de execução implementado) e o funcionamento do simulador de programas da linguagem.

4.1 Compilador

No processo de compilação de um programa RS, o compilador recebe como entrada um arquivo, que possui um ou mais módulos definindo um programa RS. Se a sintaxe estiver correta, o compilador decompõe o programa e começa a preparação da base de dados. Se houver erro de sintaxe em algum módulo, o compilador emitirá uma mensagem através do terminal e interromperá o processo de compilação do programa.

Se a verificação sintática e a carga do programa RS forem realizadas com sucesso, ficará à disposição do usuário o código objeto compilado. Este código representa um autômato finito determinístico denominado autômato RS e pode ser visualizado através do próprio terminal de usuário. A partir da ampliação realizada neste trabalho, o código pode, também, ser salvo em arquivo, que servirá como código de entrada para o novo ambiente de execução.

Além do código-fonte RS, explicado no capítulo anterior, o programa de entrada deve especificar, ao seu final, um procedimento denominado '*environment*'. Este procedimento especifica a forma de simular o programa.

Assim, depois de compilado o programa, o fluxo de controle é repassado pelo compilador ao simulador de programas.

A compilação de um programa RS é realizada através do comando *run*. Assim, digita-se "*run* a." para compilar o arquivo-fonte RS de nome 'a'. A Figura 7 ilustra o estado do ambiente RS após a compilação do arquivo 'a', que gerou 2 estados para o módulo '*timer*' e 1 para o módulo '*emitter*'.

Ao final desse processo, o controle volta para o ambiente, que aguarda por interação do usuário (indicado por "**waiting*" na Figura 7). Neste estágio, o usuário pode iniciar a simulação do programa RS, indicando os estímulos (sinais de entrada) ao sistema.

```
1 ?- run a.

COMPILING timer
 State 1
 State 2
COMPILING emitter
 State 1
ENVIRONMENT CALLED
*waiting*
|: █
```

Figura 7 - Ambiente após a compilação

4.2 Estrutura do Código

A compilação de um programa pelo compilador RS produz um código objeto, o qual se divide em duas partes interdependentes que são estudadas de maneira conjunta. A primeira parte apresenta o *autômato* propriamente dito e a segunda, as *regras de transição* simplificadas para o programa compilado, ou seja, sem as informações desnecessárias já utilizadas na geração do código.

A primeira parte do código consta das linhas que compõem o autômato finito determinístico correspondente ao código RS compilado. A palavra AUTOMATON seguida do nome do autômato inicia sua representação. O autômato pode ser visto como a representação de uma máquina de estados, conforme Mattos [MAT 2000]. A estrutura geral de uma linha do autômato tem a seguinte forma:

```
n    s    [a]
```

onde ***n*** representa o número do estado, ***s*** é o sinal de entrada e **[*a*]** é o conjunto (lista) de ações a serem executadas quando o autômato estiver no estado *n* e for estimulado externamente pelo sinal *s*, com uma conseqüente mudança de estado. Considere, por exemplo, o seguinte fragmento de um autômato RS:

```
1   ligado    [3,*,go_to(4)]
```

Esta linha indica que, quando o sistema estiver no estado **1** e receber um sinal de entrada **ligado**, a ação **3** será executada, com conseqüente mudança do estado do sistema para 4.

O código também pode representar regras de transição a serem executadas em paralelo. Os números correspondentes às regras a serem executadas em paralelo devem aparecer entre dois asteriscos consecutivos. Assim, para um código como

```
1   ligado [2,6,*,8,*, go_to(1)]
```

a semântica será: quando o autômato estiver no estado **1** e for estimulado pelo sinal **ligado**, as ações **2** e **6** serão executadas em paralelo, seguidas pela execução da ação **8**. Depois de realizadas as execuções, o autômato irá para o estado **1**.

A primeira linha do autômato sempre conterá um estado indicado por '*init*' (equivalente ao estado 0 do autômato). Este estado corresponde à realização das ações de inicialização do programa, indicadas na declaração *initially* do programa-fonte RS. Estas ações são executadas imediatamente antes do autômato entrar em funcionamento.

As ações de uma linha do autômato podem indicar, também, regras condicionais (regras *Case)*. As regras *Case* permitem que uma condição seja avaliada, indicando se seus comandos serão ou não executados. Se a condição não for verdadeira, a próxima condição será testada, e assim sucessivamente, até que a primeira condição verdadeira seja encontrada. Como já foi dito, a inexistência de uma condição verdadeira para regras condicionais implica em um erro de lógica no programa, o qual não é indicado em tempo de compilação.

As ações condicionais podem aparecer imediatamente após um asterisco, no lugar onde normalmente apareceria uma instrução do tipo *go_to*, ou até mesmo no lugar onde apareceria a primeira regra de reação. Neste último caso, a primeira ação já indicará uma regra condicional. A sintaxe para o primeiro caso equivale a:

```
1  ligado  [3,*,[5-1,*,go_to(1)],[5-2,*,go_to(1)]]
```

A semântica para o trecho de código anterior é a seguinte: quando o autômato estiver no estado **1** e for estimulado pelo sinal **ligado**, a ação **3** será executada; em seguida, avalia-se a condição expressa na regra **5-1**: se ela for verdadeira, será executada; caso contrário, avalia-se a condição de **5-2**.

No segundo caso, a sintaxe equivale a:

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

A semântica é semelhante à anterior, com a diferença de que, neste caso, somente é executada a ação referente à regra *Case*.

As ações condicionais podem aparecer aninhadas, ou seja, uma ação condicional pode referenciar outra ação de mesmo tipo, e assim por diante.

A estrutura geral do código gerado para o autômato é ilustrada pela Figura 8.

```
AUTOMATON nome_do_automato:
init      -        [x,*,go_to(z)]
1         s_entr   [a,*,go_to(b)]
```

Figura 8 - Sintaxe do Autômato RS

A segunda parte do código contém as linhas que compõem as regras de transição, as quais complementam o autômato gerado pelo compilador. Estas regras são referenciadas pelo autômato e descrevem os comandos a serem executados. A palavra RULES indica o início da segunda parte. De forma geral, uma regra de transição gerada tem o seguinte formato:

```
n. [condição] ===> [ação]
```

onde **n** representa o número (identificação) da regra a ser executada, o qual é representado no autômato; **[condição]** indica as condições (conhecidas como *condições de disparo*) que precisam ser verdadeiras para que a regra possa ser executada e **[ação]** é uma lista de comandos que serão executados quando a regra *n* for indicada no autômato e as condições forem verdadeiras. Quando houver mais de um comando, eles serão separados por vírgulas.

A *condição de disparo* pode conter sinais de entrada e/ou internos, separados por vírgulas. No máximo um sinal de entrada poderá aparecer na condição, pois um programa RS pode tratar somente um sinal de entrada por reação. Os sinais internos, contudo, não sofrem essa restrição.

A lista de comandos referida em **[ação]** pode constar de instruções de *atribuição* e comandos do tipo *emit* e *up*. Instruções de atribuição inserem em uma variável o valor de uma expressão e são denotadas por:

```
v := expressão
```

sendo *v* uma variável.

Os comandos *up* e *emit* servem, respectivamente, para ligar um sinal interno e emitir um sinal externo. Sua sintaxe é denotada por:

```
up(sinal_interno)

emit(sinal_de_saida)
```

sendo que *sinal_interno* somente pode ser valorado, visto que sinais puros são eliminados pelo compilador RS. *sinal_de_saida* pode ser um sinal puro ou valorado.

Convém salientar que tanto *condição* quanto *ação* podem ser vazias, ou seja, não é obrigatório que existam condições e comandos para uma determinada regra.

As regras representadas acima são *incondicionais*, ou seja, estando as condições de disparo habilitadas, os comandos que estiverem em **[ação]** serão executados. Todavia, regras de reação também podem ser *condicionais*. Tais regras são identificadas pela palavra *Case* e apresentam condições especiais para seu disparo.

Regras condicionais (ou comandos de seleção) permitem uma verificação sobre um conjunto de expressões de forma a decidir quais ações devem ser executadas e qual será o próximo estado do autômato. A Figura 9 mostra a forma como estas regras aparecem no código objeto, onde **{ExprCond_i}** indica uma expressão que possui um resultado booleano (true ou false) e ***else*** indica uma condição default que é executada caso nenhuma das condições anteriores seja verificada. Novamente, deve ser ressaltado que o uso da cláusula *else* é opcional e que a falta de uma condição verdadeira numa avaliação de expressão condicional configura um erro de lógica na programação, não identificada em tempo de compilação.

```
n.   Case:
n-1. [CondDisp] {ExprCond_1} ---> [Ação_1]
...
n-x  [CondDisp] {else} ---> [Ação_x]
```

Figura 9 - Sintaxe de Regras Condicionais

A estrutura geral do código gerado para as regras de transição é ilustrada na Figura 10, onde <Regra_de_Reação$_i$> pode ser uma regra condicional ou incondicional, de acordo com a sintaxe a pouco definida.

```
RULES FOR nome_do_automato:
Module nome_do_modulo:
     1.   <Regra_de_Reação1>
     .    .          .    .
     k.   <Regra_de_Reaçãok>
```

Figura 10 - Código Gerado para Regras de Transição

A Figura 11 mostra um exemplo prático de um código objeto RS. Este é o código referente ao programa rst, cujo programa-fonte foi mostrado na Figura 5, no capítulo 3.

```
AUTOMATON rst:
     init -    [1,*,go_to(1)]
     1    r    [2,*,go_to(2)]
     2    r    [2,*,go_to(2)]
     2    s    [3,*,go_to(3)]
     2    t    [4,*,go_to(1)]
     3    s    [5,*,go_to(1)]
     3    r    [2,6,*,go_to(2)]
     3    t    [7,*,go_to(1)]
RULES FOR rst:
Module rst:
     1.   []   ===> []
     2.   []   ===> []
     3.   []   ===> []
     4.   []   ===> []
     5.   []   ===> []
     6.   []   ===> []
     7.   []   ===> [emit(ok)]
```

Figura 11 - Exemplo de Código Objeto RS

4.3 Simulador

Ao final do processo de compilação, o compilador transfere o controle para o ambiente (ou "*environment*") de execução. O ambiente aguarda a intervenção do usuário para realizar alguma tarefa, que pode ser: simular o programa, depurá-lo, mostrar autômato(s) e/ou regras, ou ainda abortar o processo de simulação. Esta última opção faz com que o interpretador volte ao estado de espera pela compilação de um novo programa.

A *simulação* consiste em estimular diretamente o autômato desejado (caso houver mais de um) através de pares "<nome_do automato>-<sinal>.", sendo que os sinais de saída serão mostrados no terminal de forma textual.

A forma como os sinais de entrada e saída são enviados ou emitidos pelo autômato é regulada pelo predicado "*environment*", declarado pelo usuário ao final do código-fonte RS. Com o intuito de facilitar a tarefa de programação, são oferecidos dois procedimentos padrões: "*user_terminal*" e "*cascade*". Dessa forma, o usuário opta em seu código por utilizar "*environment* :- *user_terminal*" ou "*environment* :- *cascade*".

O predicado "*user_terminal*" permite estimular os autômatos através do teclado, sendo os sinais de saída mostrados diretamente no terminal de usuário. Neste caso, o estímulo deve ser um par "<nome_do_automato>-<sinal>".

O predicado "*cascade*" é similar a "*user_terminal*", porém as saídas de um autômato que sejam entradas para outros autômatos são usadas para (imediatamente) estimulá-los e não saem para o exterior (no caso, o terminal).

Para evitar o uso de "<nome_do_automato>-<sinal>" no processo de estimulação de autômatos, pode-se fixar um autômato "p" para uma seqüência de interações, através do predicado "to(p)". Neste caso, os sinais para "p" não precisarão ser prefixados com o nome do autômato.

Além de simular, é possível realizar a *depuração* de um programa através do predicado mk(msg). Este predicado é um instrumento de depuração que, quando chamado, interrompe a execução do programa e disponibiliza uma janela de inspeção, através da qual pode-se examinar o estado do sistema.

A janela aberta por mk(msg) permite ao usuário entrar com qualquer comando Prolog. Após a execução de cada comando, o usuário pode optar entre continuar a inspeção ou retornar à execução normal (através do comando "c."), até a próxima pausa. O comando "*b_off*" (ou "*go*") desativa as pausas e o comando "*b_on*" reativa-as. O argumento "msg" pode ser usado para identificar o ponto onde a janela foi aberta.

Além disso, o interpretador RS dispõe de alguns predicados adicionais bastante úteis. Tais predicados são: “sa” (show_a), “sr”(show_r), e “sar”(show_ar). O comando “sa” permite a visualização do(s) autômato(s) equivalente(s) ao programa compilado. Assim, digitando “sa.”, o(s) autômato(s) armazenado(s) na base de dados do Prolog será(ão) mostrado(s) ao usuário na tela do terminal. Semelhantemente, o mesmo ocorre para “sr.”, que mostra as regras de reação do programa, e para “sar.”, que mostra, simultaneamente, o(s) autômato(s) e as regras do programa compilado. Convém salientar que as regras de reação apresentadas já estão simplificadas, ou seja, não aparecem informações desnecessárias já utilizadas na geração de código (mais especificamente, sinais internos puros são eliminados do código apresentado por não conduzirem dados). A Figura 12 ilustra a saída do comando “sar.” na tela do terminal Prolog.

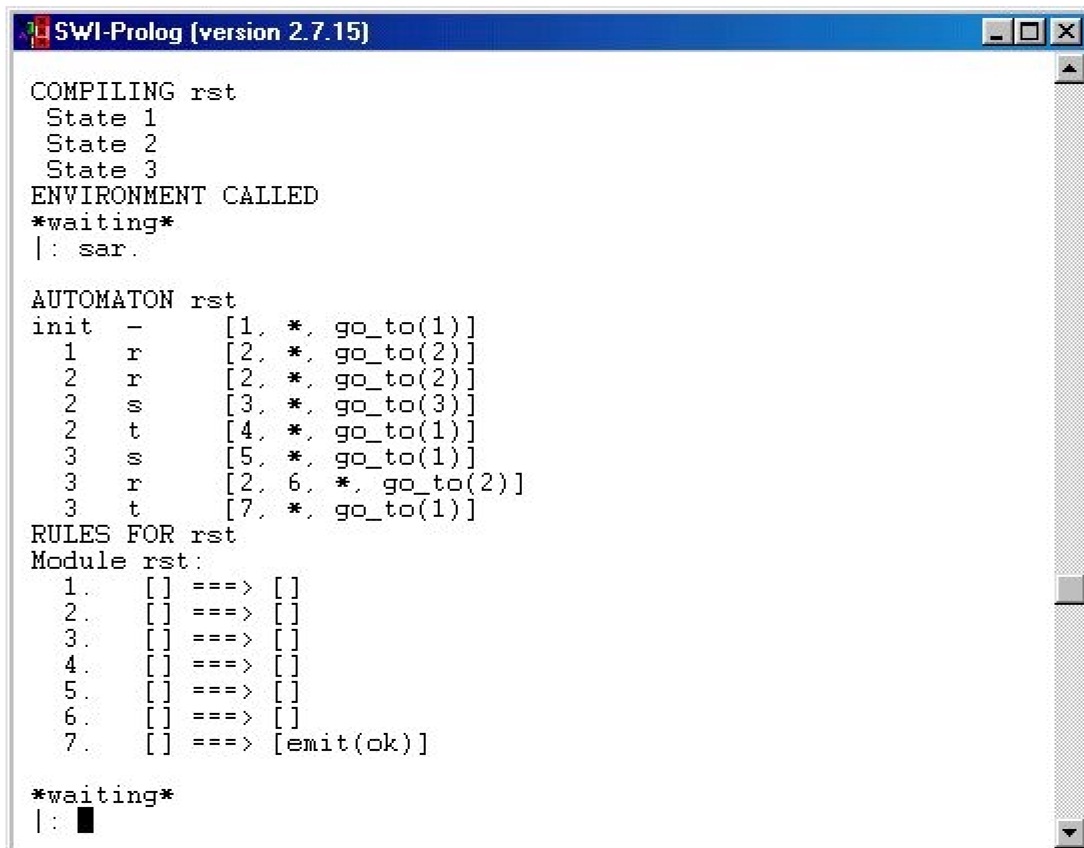

Figura 12 - Utilização do comando 'sar'

5 Considerações sobre interfaces de usuário

Este capítulo contempla as partes mais relevantes sobre interfaces de usuário no contexto utilizado neste trabalho. São abordados aqui alguns conceitos referentes a interfaces, além de aspectos sobre usabilidade, comunicabilidade e estilos de interação com o usuário, com o intuito de demonstrar a necessidade de uma renovação na interface do ambiente RS original.

5.1 Definição de Interface

Segundo Souza [SOU 99], pode-se definir o termo *interface* como "aquilo que interliga dois sistemas". Segundo Moran (apud [SOU 99]), "a interface de usuário deve ser entendida como a parte de um sistema computacional com a qual uma pessoa entra em contato física, perceptiva e conceitualmente".

É igualmente possível considerar uma interface homem-máquina como o mecanismo que permite ao usuário controlar e avaliar o seu funcionamento de um determinado artefato através de dispositivos que estimulem a sua percepção.

No contexto relativo ao processo de interação usuário-sistema, a interface é um combinado de software e hardware necessário para viabilizar e facilitar os processos de comunicação entre o usuário e a aplicação. Considera-se, então, a interface como sendo um sistema de comunicação, uma vez que oferece os instrumentos necessários para os processos de comunicação e de interação entre o usuário e o sistema.

A definição de Moran para interfaces de usuário permite caracterizá-las como possuindo um componente físico (ou de *hardware*), o qual é percebido e manipulado pelo usuário, e um outro conceitual (ou de *software*), que o usuário interpreta e processa.

Os *componentes de hardware* de uma interface agregam os dispositivos com os quais os usuários realizam as atividades motoras e perceptivas, ou seja, dispositivos que o usuário pode efetivamente tocar e manipular. Entre os componentes mais amplamente conhecidos estão o monitor de vídeo, o teclado e o mouse.

O *componente de software* da interface é responsável, basicamente, por implementar os processos computacionais necessários para:

a. controlar os dispositivos de hardware;

b. construir dispositivos virtuais com os quais o usuário pode interagir;

c. gerar os diversos símbolos e mensagens que representam as informações do sistema;

d. interpretar os comandos indicados pelo usuário através dos componentes de hardware controlados.

De um modo geral, as interfaces entre usuários e sistemas computacionais diferenciam-se das utilizadas por máquinas convencionais pelo fato de requisitarem do usuário um maior esforço na interpretação das informações que o sistema processa segundo Norman (apud [SOU 99]).

5.2 Processo de Interação

O processo de interação engloba as ações do usuário realizadas sobre a interface de um sistema e suas interpretações sobre as respostas reveladas por esta interface. Esta interação pode ser facilitada ou dificultada de acordo com os níveis de usabilidade e comunicabilidade que a interface apresenta.

A usabilidade de um sistema pode ser vista como a qualidade da interação do sistema com os usuários. Ela pode ser medida levando-se em conta vários aspectos, dentre os quais pode-se citar: facilidade de aprendizado do sistema, facilidade de uso, satisfação do usuário, flexibilidade e produtividade.

A comunicabilidade de um sistema é a sua propriedade de transmitir ao usuário, de forma eficaz e eficiente, as intenções e princípios de interação que guiaram o seu desenvolvimento. Um sistema que possui uma boa comunicabilidade é capaz de mostrar ao usuário, durante o processo de interação com a interface, as razões pelas quais determinadas decisões no projeto da interface foram tomadas. Juntas, a usabilidade e a comunicabilidade pretendem aumentar a aplicabilidade de um software.

Outra característica importante de uma interface diz respeito à facilidade com que ela revela aos usuários suas propriedades percebidas e reais, em particular aquelas que determinam a maneira pela qual ela pode ser utilizada. Esta propriedade é referida, segundo Norman, pelo termo *affordance*. A plena utilização desta característica faz com que o usuário da interface consiga perceber precisamente as ações a realizar através dela

tão somente olhando-a. Por exemplo, a *affordance* de um botão é que ele seja pressionado.

Segundo Preece e Shneiderman (apud [SOU 99]), as formas de comunicação ou interação entre os usuários e os sistemas computacionais definem o que se chama de *estilo de interação*. Dentre os vários estilos de interação existentes, podem ser destacados, para este trabalho, os seguintes: *linguagens de comando* e *WIMP* (*Windows, Icons, Menus and Pointers*).

As interfaces baseadas em *linguagens de comando* proporcionam ao usuário a possibilidade de enviar instruções diretamente ao sistema através de comandos específicos. Estas linguagens são consideradas poderosas por oferecerem acesso direto à funcionalidade do sistema. Além disso, elas permitem uma maior iniciativa do usuário e uma maior flexibilidade na construção dos comandos, através da variação de parâmetros e da combinação de palavras e sentenças. Contudo, este poder e flexibilidade implicam uma maior dificuldade dos iniciantes em aprender e utilizar o sistema. Os comandos e a sintaxe da linguagem precisam ser relembrados e erros de digitação são comuns mesmo nos mais experientes.

Diferentemente do anterior, o estilo de interação *WIMP* (acrônimo inglês para Janelas, Ícones, Menus e Apontadores) permite a interação através de componentes de interação virtuais denominados *widgets*. Este estilo utiliza as tecnologias de interfaces gráficas, proporcionando o desenho de janelas e o controle de entrada através do teclado e do mouse em cada uma destas janelas. É possível, através deste estilo, utilizar menus fixos e de contexto, botões, seletores de opções e diversos outros dispositivos virtuais com o intuito de facilitar a interação entre o usuário e o software.

Em geral, o uso estilo de interação WIMP em detrimento do anterior facilita o uso do software pelo usuário, sobretudo para os iniciantes, porém diminui a flexibilidade do sistema, que é um dos pontos fortes do estilo baseado em linguagens de comando. A partir dessa constatação, fica bem caracterizada a importância de modelar e implementar um novo ambiente de execução para a linguagem RS que efetivamente utilize as facilidades referidas no estilo WIMP. A modelagem e implementação desta nova abordagem de interface para a linguagem RS podem ser vistas, respectivamente, nos capítulos 6 e 8, a seguir.

6 O Novo Ambiente de Execução

Os capítulos anteriores ilustraram o formato original do ambiente de execução RS. Através deles, foi possível observar a necessidade de elaboração um novo ambiente que pudesse melhorar a interação usuário-sistema.

Este capítulo mostra os principais aspectos de projeto do novo ambiente de execução, que vão desde a atualização do interpretador RS, passando pela definição da máquina virtual, até a modelagem da interface visual.

6.1 Aspectos Gerais de Projeto

A Figura 13 ilustra, de maneira geral, o esquema de projeto proposto para o novo ambiente de execução implementado neste trabalho.

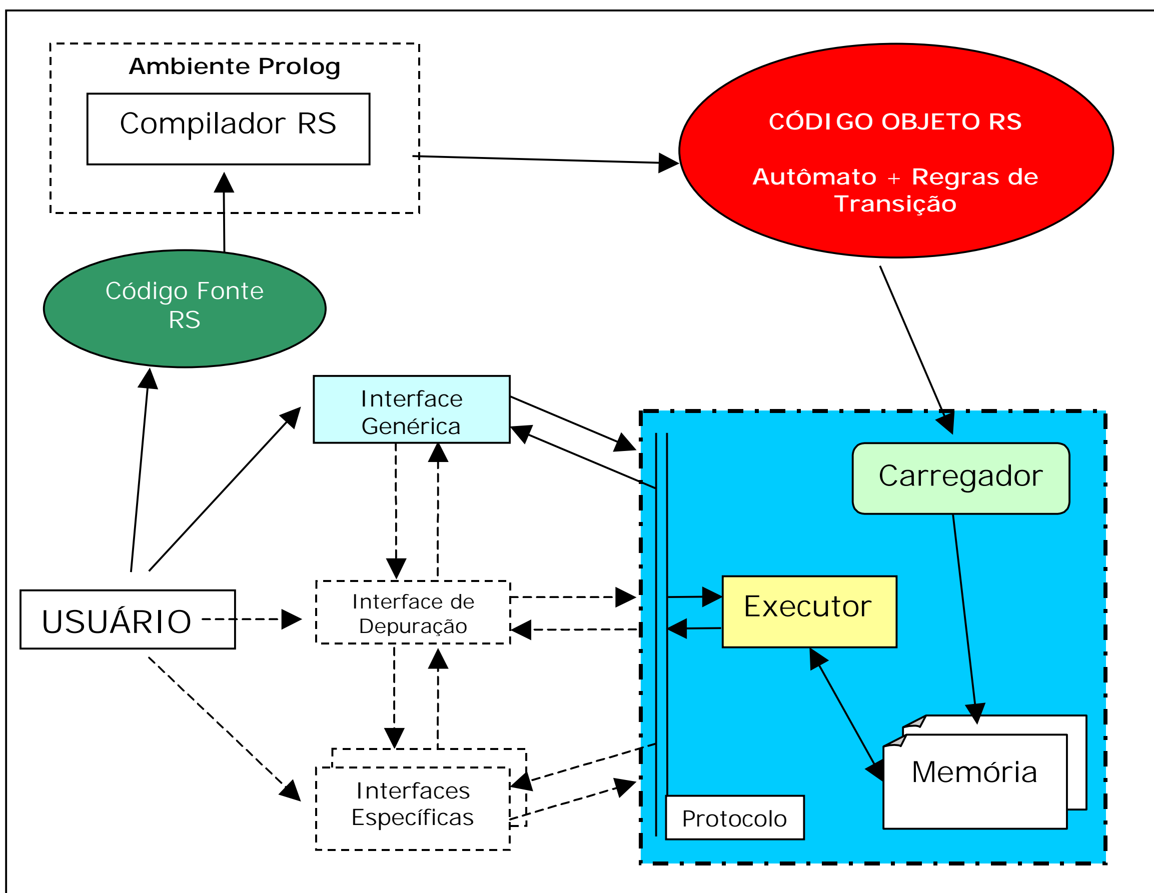

Figura 13 - Esquema de projeto do novo ambiente de execução

Como pode ser visto, o código fonte da linguagem RS passa por um processo de compilação, gerando desta forma o código objeto que é utilizado como entrada no novo ambiente de execução. Este código possui duas partes (autômato e regras de transição), compondo um módulo que é armazenado em um arquivo na forma de texto conforme formato detalhado na seção 4.2. É possível a existência de programas constituídos por mais de um módulo. Entretanto, no presente trabalho, para prototipar o novo ambiente de execução, são considerados válidos somente os programas compostos por um só módulo, conforme a versão RS-0 da linguagem.

Para que este código pudesse estar disponível na forma de arquivo, foi necessária uma pequena modificação no compilador com o intuito de permitir a gravação do código em arquivo.

De acordo com o esquema, o novo ambiente possui um módulo de execução composto por um **carregador**, uma **memória** e um **executor** propriamente dito. O carregador recebe o código objeto e o coloca na memória do executor. O executor se encarrega de executar o programa contido em sua memória, de acordo com os sinais de entrada fornecidos pela **interface** do usuário, como será visto posteriormente.

Além disso, uma **interface visual genérica** para a execução de programas é apresentada ao usuário em função do programa carregado. A interface, montada a partir de informações contidas no texto do programa, faz a comunicação entre o ambiente externo (no caso, o usuário) e o executor, e vice-versa. A partir dela, é possível simular um programa e visualizar os resultados da simulação executada.

Existe, ainda, uma **interface básica de depuração** baseada na interface visual genérica, a qual permite ao usuário a execução controlada de um programa, de maneira a inspecionar o correto funcionamento do mesmo através do conteúdo de suas variáveis e sinais. Adicionalmente, com a definição de um **protocolo de interação interface-executor**, é possível elaborar novas interfaces para aplicações específicas definidas pelo usuário, como o relógio digital de múltiplas funções apresentado por Toscani [TOS 93].

As seções seguintes apresentam, de forma mais detalhada, os elementos modelados.

6.2 Atualização do Interpretador RS

Através da Figura 13, é possível observar que a idéia fundamental do novo ambiente de execução é utilizar como entrada para o processo de execução o código objeto gerado pelo compilador RS. A partir desse código, o novo executor deve ser

capaz de retirar as informações necessárias para simular o programa RS. Analisando o código-fonte do interpretador da linguagem em sua versão 41, foi constatada necessidade de alterá-lo, tendo em vista a falta de um comando específico que pudesse realizar a tarefa de colocar em arquivo o código objeto RS. No intuito de resolver este problema, foram definidos os novos comandos *sv* e *sv(X)* no interpretador da linguagem RS.

Conforme Palazzo [PAL 97], durante a execução de um programa Prolog, somente dois arquivos estão ativos: um para entrada e outro para saída. Tais arquivos são denominados, respectivamente, 'fonte de entrada corrente' e 'fonte de saída corrente'. No início da execução, as duas fontes correspondem ao terminal do usuário. A qualquer momento, porém, ambas podem ser mudadas para um outro arquivo qualquer através dos objetivos Prolog *see(X)* e *tell(X)*, que correspondem, respectivamente, aos comandos para modificar a entrada e a saída padrão, sendo X o nome do arquivo em questão (X = 'user' indica terminal de usuário).

De acordo com a análise prévia do interpretador RS, foi contatado que, apesar de não possuir um mecanismo específico para gravar em arquivo o código objeto RS, o mesmo já possuía um comando para mostrar, no terminal de usuário Prolog, o conteúdo do autômato e das regras de transição para um programa RS compilado. Este comando é denominado de *sar* (acrônimo da expressão *Show Automaton and Rules*).

A partir da existência deste comando, foram definidos os novos comandos *sv* e *sv(X)*. Estes comandos são implementados meramente pelo redirecionamento da saída padrão do ambiente Prolog para o arquivo especificado (no caso de *sv*, para o arquivo "codObj.roc") e subseqüente utilização do comando '*sar.*' para mostrar a representação do código compilado. Dessa forma, o conteúdo gerado por *sar* é gravado em arquivo, na forma de texto, ao invés de ser mostrado no terminal de usuário.

A partir da definição destes novos comandos, o esquema de compilação de RS foi levemente alterado para realizar o salvamento automático no arquivo 'codObj.roc' do código gerado pelo compilador RS, toda vez que um programa for compilado.

Com esta modificação, existem agora quatro formas de se obter o código objeto de um programa RS: um através do salvamento automático do código gerado após a compilação do programa, outros dois através do uso dos comandos acima especificados, além da visualização do código pelo simples uso do comando *sar*.

A seqüência completa para geração do código objeto necessário ao novo ambiente é a seguinte:

- Ativação do ambiente Prolog;
- Carga do interpretador RS no Prolog;
- Utilização do comando *run* para compilar um programa fonte RS (e deixá-lo em modo de execução);
- Chamada de 'sv(X).' para efetuar o salvamento do código com um nome de arquivo específico, se desejado;
- Chamada de 'abort.' para abortar execução do programa RS ou 'halt.' para abortar execução do programa RS e fechar o Prolog.

6.3 O Módulo de Carga / Execução

O ambiente executor original da linguagem RS é composto por um compilador e um simulador com facilidades de depuração de programas textuais. O processo de compilação monta as bases de dados para o programa sendo compilado, deixando estas informações prontamente disponíveis para uso do simulador.

O código fornecido pelo compilador não disponibiliza nenhum tipo de estrutura que agregue todas as variáveis, sinais e campos utilizados, para facilitar o seu reconhecimento. Assim, para que se possa executar um programa RS no novo ambiente a partir desse código, é necessário um processo prévio de análise sobre seu texto e de carga das informações obtidas para estruturas de dados apropriadas.

Tendo como base o modelo de código objeto a ser usado como entrada para o novo ambiente de execução, o módulo de execução foi estruturado de forma a ser composto por três partes, a saber: um **carregador-tradutor**, uma **memória** e um **executor** propriamente dito. A Figura 13 ilustra essa configuração.

O *carregador* desempenha o papel de um tradutor do texto correspondente ao código gerado pelo compilador. Ele é responsável por realizar as etapas de análise léxica, sintática e semântica, colocando as informações recolhidas nesse processo em estruturas de dados definidas na memória do sistema, que é utilizada pelo executor. A carga do programa é feita num só passo. As etapas acima citadas estão baseadas na gramática definida por Mattos [MAT 99], que sofreu algumas alterações para este trabalho. O processo de análise referido é bastante semelhante àqueles descritos em livros especializados sobre o assunto [AHO 88] [PRI 2001]. Convém salientar que, dentre os processos clássicos de análise sintática disponíveis, optou-se por implementar a Análise *Preditiva Tabular*, baseada no uso de uma pilha sintática explícita e de tabela preditiva .

As estruturas de dados, que mantém a representação interna do código reconhecido pelo carregador, podem ser vistas como uma implementação da *memória* utilizada pelo executor para simular o programa.

A *memória* do executor é composta, basicamente, por quatro 'tabelas':

- uma Tabela de Variáveis, onde são armazenadas todas as variáveis utilizadas no programa;
- uma Tabela de Sinais, que comporta todos os sinais e campos de sinais do programa;
- uma Tabela de Transições do Autômato, contendo todas as linhas da parte de código correspondente ao autômato;
- uma Tabela de Regras de Transição, que armazena todas as regras encontradas na segunda parte do código objeto RS.

Existem, ainda, outras tabelas utilizadas somente durante o processo de carga-tradução (mais precisamente, uma Tabela de Símbolos, que contém todos os símbolos recolhidos no processo de análise léxica, uma Tabela de Terminais e outra de Não-Terminais, além de uma Tabela Preditiva, referida anteriormente), as quais não serão pormenorizadas aqui.

Além do carregador e da memória, existe ainda um executor, responsável por tomar o sinal proveniente da interface e, baseado no conteúdo armazenado em sua memória, realizar as reações previstas no código. Este executor estabelece um canal de comunicação com a interface que o utiliza, recebendo e enviando informações através de um protocolo especialmente definido para esta finalidade, o qual será descrito mais à frente.

O módulo executor, juntamente com as estruturas que implementam o módulo de memória, podem ser vistos como uma espécie de 'Máquina Virtual' e o módulo carregador como um programa básico de ativação da máquina (*reset*), implementado em "*firmware*" dessa máquina.

6.4 A Interface Visual

Um dos principais motivos, senão o principal, para o desenvolvimento deste trabalho foi a existência de uma interface pouco amigável entre o usuário e o ambiente de execução original da linguagem RS. Tomando por base os capítulos 4 e 5, fica claro que o ambiente de execução RS original não é agradável no que diz respeito a sua utilização por estar baseado no estilo de interação de linguagens de comando (conforme

capítulo 5). Justamente a falta de uma interface amigável, que facilite a interação entre usuário e o próprio ambiente de simulação, dificulta o uso do sistema.

Para o novo ambiente de execução, procurou-se melhorar este aspecto através do uso de uma visão diferente de interface. Com a escolha do modelo de interação WIMP (*Windows, Icons, Menus and Pointers*), que utiliza janelas, botões, menus e outros itens visuais (ver seção 5.2), o usuário tem a sua disposição uma maneira de interagir com o sistema mais amigável e intuitiva, que o leva a descobrir as potencialidades do sistema por si mesmo, com o auxílio da própria interface.

Analisando as interações entre o sistema e o usuário, facilmente observa-se que a interface visual pode ser dividida em duas grandes partes: uma *interface de carga* e uma *interface de execução*. Cada uma delas possui características bastante particulares, sendo que toda a comunicação entre a interface e o módulo executor deve ser baseada no uso de um protocolo de comunicação específico definido de forma a facilitar o desenvolvimento de novas interfaces. Esse protocolo será detalhado na seção 8.2.

A interface de carga deve implementar mecanismos para configuração da máquina virtual e para visualização do conteúdo de sua memória e erros ocorridos no processo de carga, além dos comandos necessários para a seleção, carga e execução de programas RS.

A interface de execução deve implementar mecanismos que realizem a entrada de sinais (e dados associados) para o programa carregado, a visualização de resultados (sinais de saída), bem como a definição de modos de operação. Fundamentalmente, dois modos de operação devem ser previstos:

a. um *modo interativo*, que corresponde ao modelo de interação originalmente utilizado no ambiente de execução inicialmente desenvolvido para a linguagem. Neste modo, o usuário indica individualmente cada sinal de entrada, no momento desejado, durante a execução do programa, especificando valores para campos de entrada, se houver;

b. um *modo programado*, em que o usuário realiza uma ‘programação’ prévia dos sinais que devem ser repassados ao executor e dos respectivos instantes de disparo desses sinais, antes de iniciar a execução do programa.

7 Implementação do Módulo Executor

O módulo executor, encarregado de realizar a carga e execução de um programa RS, pode ser dividido em três partes: uma *memória*, que abriga os dados recolhidos do código pelo carregador; um *carregador*, responsável pela carga do código para a memória; e um *executor*, que é responsável por realizar as reações, de acordo com o sinal de entrada recebido e com o código contido na memória. No contexto deste trabalho, dentre as três partes citadas, pode-se dar um destaque maior para o *executor* pelo fato de ser o responsável pela realização das reações, ou seja, ele concentra a funcionalidade do sistema. A divisão do módulo executor pode ser vista na Figura 14.

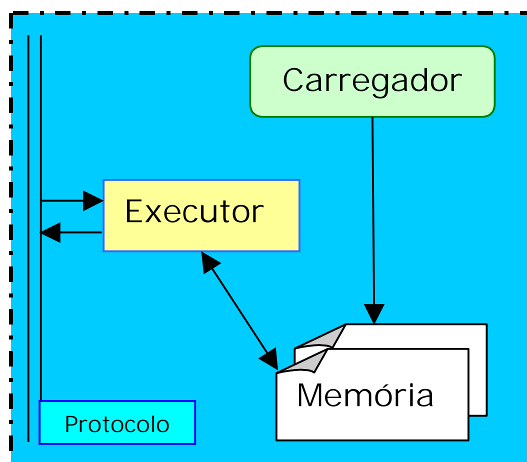

Figura 14 - Diagrama do módulo executor

Este capítulo mostra como foi feita a implementação deste módulo. No desenvolvimento do novo ambiente foi utilizada a linguagem Object Pascal através da ferramenta de programação Delphi, da Borland. Alguns testes com o interpretador RS original foram realizados utilizando o ambiente de programação SWI-Prolog, o qual é de livre utilização.

7.1 A Memória

A memória é composta basicamente pelas estruturas responsáveis por armazenar os dados recolhidos do código objeto pelo carregador, que serão utilizados posteriormente pelo executor. Ela foi dividida em quatro tabelas distintas, que são:

tabela de variáveis, tabela de sinais, tabela de transições do autômato e tabela de regras de transição. A Figura 15 ilustra a estrutura da memória.

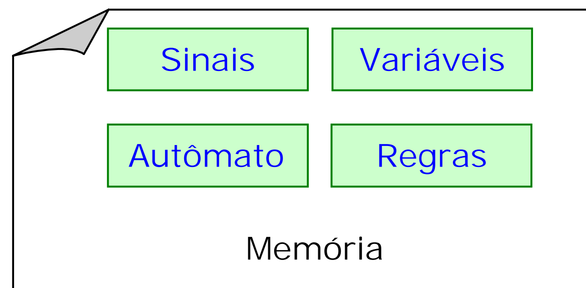

Figura 15 - Estrutura da memória

A **tabela de variáveis** é composta por uma lista encadeada de nodos, cada um contendo informações sobre uma das variáveis encontradas no código fonte. Tais informações são, basicamente, o *nome* e o *valor* da variável num determinado instante de execução. Durante a carga do programa, todas as variáveis (que podem conter somente valores numéricos, reais ou inteiros) recebem 0 como valor inicial. A Figura 16 ilustra a estrutura desta tabela.

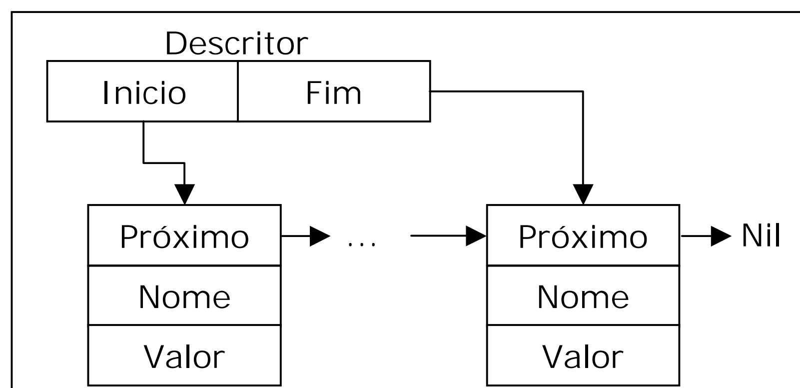

Figura 16 - Estrutura da Tabela de Variáveis

A **tabela de sinais** contém todos os sinais (tanto sinais de entrada quanto sinais internos e de saída) encontrados no programa e seus respectivos campos de valor. A estrutura desta tabela é uma lista encadeada constando dos seguintes itens em cada nodo: *nome*, *tipo*, *valor* e um ponteiro para uma *lista de campos de valor*, dado que o número de campos para um sinal é variável. A Figura 17 ilustra seu formato.

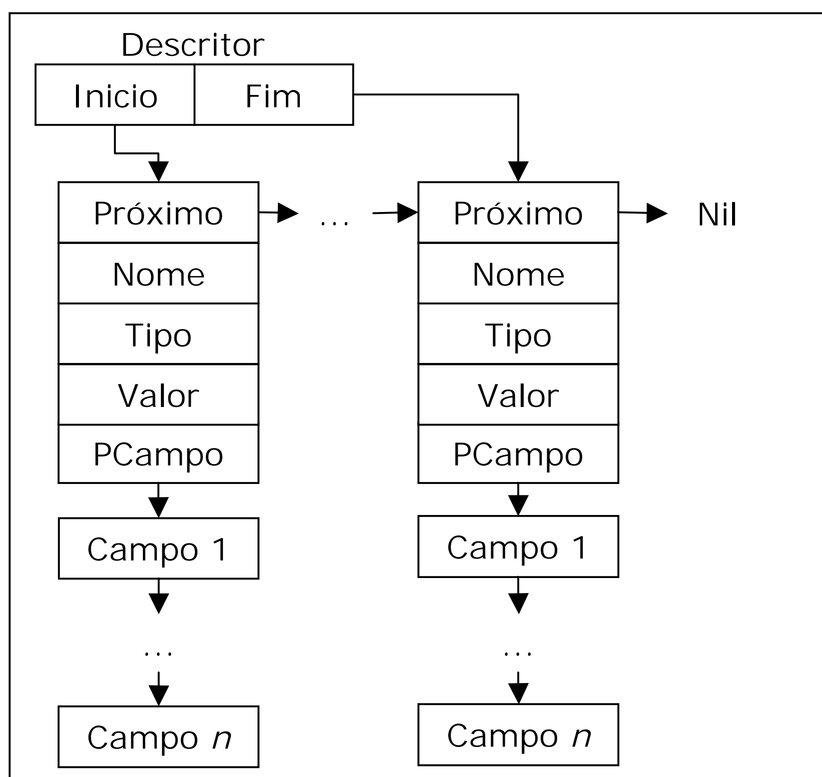

Figura 17 - Estrutura da Tabela de Sinais

A **tabela de transições do autômato**, cuja estrutura pode ser vista através da Figura 19, armazena cada uma das linhas do autômato. A forma como as informações estão dispostas numa linha do autômato (vide seção 4.2) foi determinante para a elaboração da estrutura de dados que as contém. Assim, a tabela como um todo é composta por um array de ponteiros para objetos e cada uma das linhas da tabela é formada por uma estrutura que possui os seguintes campos: *estado*, *sinal de entrada*, *transição* e *regras de ação*(a serem executadas quando esta transição for realizada). A estrutura desta tabela e a forma de reconhecimento de suas informações a partir do código objeto RS é explicada em detalhes na seção 7.2.1.

Semelhantemente à tabela de transições, a **tabela de regras de transição**, cuja estrutura é representada pela Figura 20, retém todas as regras encontradas no arquivo de código processado pelo carregador. A estrutura de dados responsável por seu armazenamento é bastante complexa, sendo composta pelos seguintes campos: *número da regra*, um ponteiro para uma *lista de parâmetros*, um ponteiro para uma *lista de condições* e um ponteiro para uma *lista de ações*. A seção 7.2.2 provê maiores informações a respeito da estrutura desta tabela e do processo de reconhecimento de suas informações.

Cada uma das tabelas é implementada como uma classe, de forma que, para se ter acesso ao seu conteúdo, tanto para leitura quanto para escrita, é preciso fazer uso dos métodos responsáveis por tal tarefa. Cada uma das classes que definem uma tabela possui métodos próprios e específicos.

7.2 O Carregador

O carregador é responsável por transferir todos os dados relevantes encontrados no código objeto para a memória da máquina virtual, conforme descrito na seção anterior. Assim, o novo executor implementado faz uso efetivo do conteúdo armazenado na memória para realizar a tarefa de execução de reações.

A máquina virtual, através do carregador, recebe como entrada um arquivo contendo código gerado pelo compilador RS. Todo código oriundo de um processo de compilação é considerado sintaticamente correto. Nada, porém, impede o usuário de ‘montar’ seu próprio código objeto ‘a mão’ e submetê-lo para carga, conhecendo-se a sintaxe. Além disso, o código objeto RS fornecido pelo compilador vem em seu estado ‘puro’, ou seja, toda e qualquer informação para sua execução deve ser extraída de seu código.

Diante destes fatos, optou-se por submeter o código de entrada a um processo de análise (léxica, sintática e semântica) para, ao mesmo tempo, garantir a correção sintática do código e realizar o processo de recolhimento das informações necessárias para sua simulação (autômato, regras de reação, sinais, variáveis, etc). A Figura 18 ilustra o processo de carga de um arquivo objeto RS para a memória.

Através da figura, é possível observar os principais passos realizados para o reconhecimento do código RS. Primeiramente, o *scanner* (Analisador Léxico) recebe o arquivo de entrada e envia, quando solicitado, os *tokens* encontrados para o *parser* (Analisador Sintático) e para a Tabela de Símbolos. O *parser*, com o auxílio da Tabela Preditiva e da Pilha Sintática, reconhece (ou não) as estruturas compostas por seqüências de *tokens* vindos do *scanner*. Em ocasiões especiais, o analisador sintático dispara o analisador semântico para que ele coloque as informações encontradas na memória, fazendo uso da Tabela de Símbolos montada.

Para a elaboração do analisador sintático, tomou-se por base a gramática para o código objeto RS, definida por Mattos [MAT 99]. Esta gramática foi alterada para atender às necessidades específicas deste projeto (foram realizadas etapas de fatoração e eliminação de recursividade à esquerda, por exemplo). Estas modificações tornaram

possível a utilização do método de *Análise Preditiva Tabular* [AHO 88] [PRI 2001] para reconhecimento de código. O analisador construído com base nesse método faz uso de uma pilha sintática explícita, ao invés de chamadas recursivas e uso de pilha implícita, como na *Análise Recursiva Preditiva*.

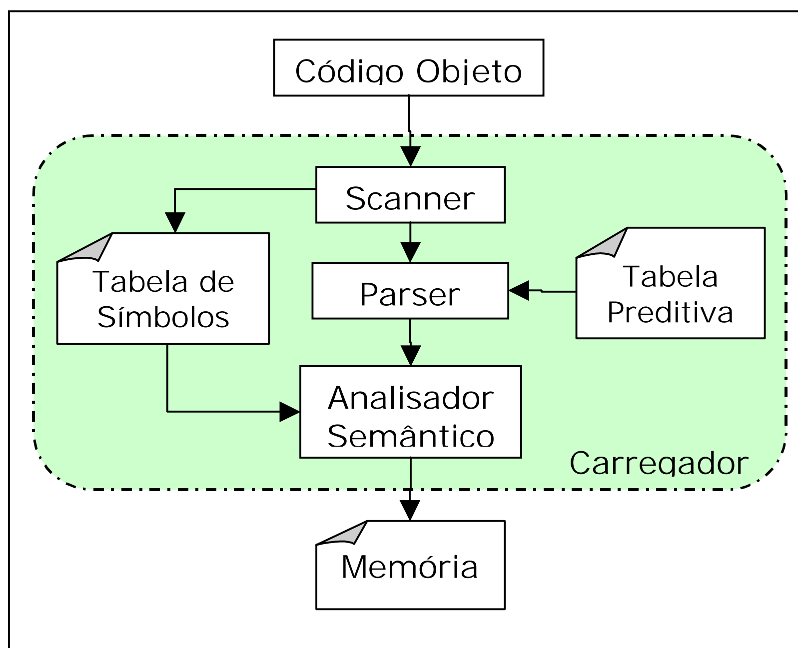

Figura 18 - Esquema de Carga para Memória

Conforme Price [PRI 2001], o princípio do reconhecimento preditivo é a determinação da produção da gramática que deve ser aplicada, levando-se em conta o símbolo não-terminal no topo da pilha e o *token* lido na entrada pelo *scanner*. O *parser* deve buscar, na tabela de análise, a produção a ser aplicada, substituindo o terminal no topo da pilha sintática pelo lado direito da produção encontrada.

Na montagem da tabela preditiva de análise, utilizada pelo analisador sintático, foram determinados o Conjunto dos Primeiros (*First*) e o Conjunto dos Seguintes (*Follow*) através do uso de algoritmos específicos existentes na literatura especializada [AHO 88] [PRI 2001].

Da mesma forma que o código fornecido pelo compilador é dividido em duas partes distintas, porém complementares, o processo de reconhecimento do mesmo também foi dividido em duas partes, a saber: *reconhecimento do autômato* e *reconhecimento das regras de transição*. A seguir, cada um dos processos de reconhecimento será visto em detalhes.

7.2.1 Reconhecimento e Armazenamento do Autômato

Para o armazenamento do autômato, foi definida uma tabela (*array*) dinâmica, chamada **Tabela de Transições do Autômato**, cuja estrutura pode ser vista na Figura 19.

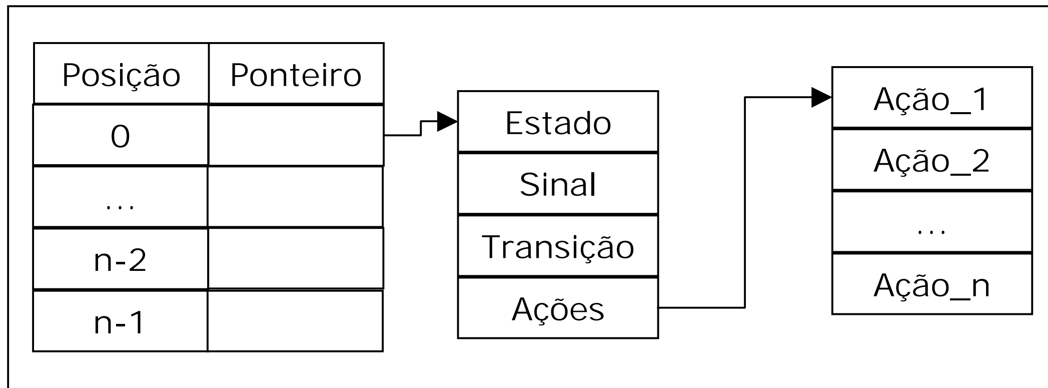

Figura 19 - Esquema da Tabela de Transições do Autômato

O campo *estado* armazena um número de estado possível para o autômato e o campo *sinal* armazena o nome de um sinal externo que pode estimular o autômato.

O campo de *transição* pode assumir dois tipos de valores: um *numérico*, que indica o próximo estado do programa, e outro *não-numérico*. O valor não-numérico caracteriza a existência de uma regra condicional a ser executada para a determinação do próximo estado do programa. Ele também fornece a posição dentro desta mesma tabela onde a regra condicional (*case*) está armazenada. Dessa forma, ao encontrar um valor não-numérico, o autômato faz uma espécie de *transição-falsa*: ao invés de obter o próximo estado de maneira direta, é feito um desvio para a execução de uma outra regra e posterior obtenção do novo estado.

O campo de *ações* é composto por um vetor dinâmico de *bytes*, ou seja, um vetor cujo tamanho é variável e dependente do número de regras de transição a serem executadas naquela linha. Devido a uma convenção adotada em termos de projeto, esse vetor possui o dobro do tamanho necessário para seu armazenamento.

Cada uma das identificações para ações normais encontradas numa linha do autômato poderia ocupar somente um *byte* no vetor. Porém, as regras de ação do tipo *case* sempre são identificadas por pares de valores (como 5-1, 5-2, e assim por diante), indicando o número da regra e a ação a ser tomada. Definiu-se, então, que as ações normais serão sempre acrescidas de um byte de *flag* com um valor zero para diferenciá-las das ações condicionais (*case*).

Assim, todas as ações encontradas ocupam 2 bytes na memória. Para as regras normais (não-condicionais), o primeiro *byte* contém o número da regra e o seguinte, um valor *zero* como *flag*. Nas regras do tipo case, o primeiro *byte* armazena a parte principal de seu identificador e o segundo *byte* indica a opção a ser avaliada.

As ações condicionais que aparecerem em uma linha do autômato são, também, armazenadas na tabela de transições do autômato, porém ao final desta. Apesar de armazenadas ao final da tabela, essas regras estarão dispostas em ordem crescente de aparecimento. Ou seja, primeiro aparecerá a regra condicional correspondente à primeira condição, depois à segunda, e assim sucessivamente. Na linha em que aparecer a referência à regra *case*, o campo de transição conterá, ao invés de um número de estado, um valor especial que indica a posição, nessa mesma tabela, a partir do qual as regras do *Case* indicado estarão armazenadas, representando um comportamento semelhante a um comando de desvio.

O armazenamento de uma regra de ação condicional tem a seguinte lógica: o campo de estado conterá o número principal do *case*, o campo de sinal conterá o valor de indexação das condições, indicando a ordem (de menor à maior) para aquela ação e os demais campos conterão seus valores normais.

Se uma regra case fizer referência a outra regra condicional, seu campo de transição armazenará o índice na tabela onde a regra referenciada for alocada, da mesma forma como foi explicado anteriormente.

Na tabela de transições, todas as linhas estarão armazenadas em ordem crescente de número de estado (exceção feita às regras condicionais armazenadas ao final da tabela). Assim, primeiro aparecerá a linha que contém estado *init* (inicial), depois as linhas contêm o estado 1, seguidas pelas que contêm o estado 2, e assim sucessivamente.

Para um determinado estado, podem existir uma ou mais regras de reação alternativas a disparar, de acordo com o sinal de entrada. Este padrão de comportamento permitiu a indexação externa desta tabela. Através de um vetor especial, é possível informar com precisão onde inicia dentro da tabela de transições as opções correspondentes a um determinado estado, de forma a agilizar o processo de execução do programa.

Em meio ao reconhecimento das linhas do autômato, a tabela de sinais vai sendo parcialmente ocupada pelos sinais indicados nesta parte do código. Nesta fase, são obtidos somente os nomes dos sinais de entrada, visto que ainda não é possível

determinar a existência de campos para estes sinais. Dessa forma, aqueles sinais que contiverem campos de valor permanecerão armazenados de maneira incompleta.

7.2.2 Reconhecimento e Armazenamento das Regras de Transição

O reconhecimento das regras de transição é uma tarefa complexa que se segue imediatamente depois de efetuado o reconhecimento das linhas do autômato. Nesta fase, o código é reconhecido, de maneira semelhante ao anterior, e as tabela de variáveis, de sinais e de regras são preenchidas com os itens encontrados no processo de análise.

O armazenamento das regras de transição é feito através de uma **Tabela de Regras de Transição**. Esta tabela é composta por um *array* dinâmico de ponteiros para uma estrutura de dados que armazena todos os elementos de uma regra. O modo como os dados são armazenados é apresentado através da Figura 20.

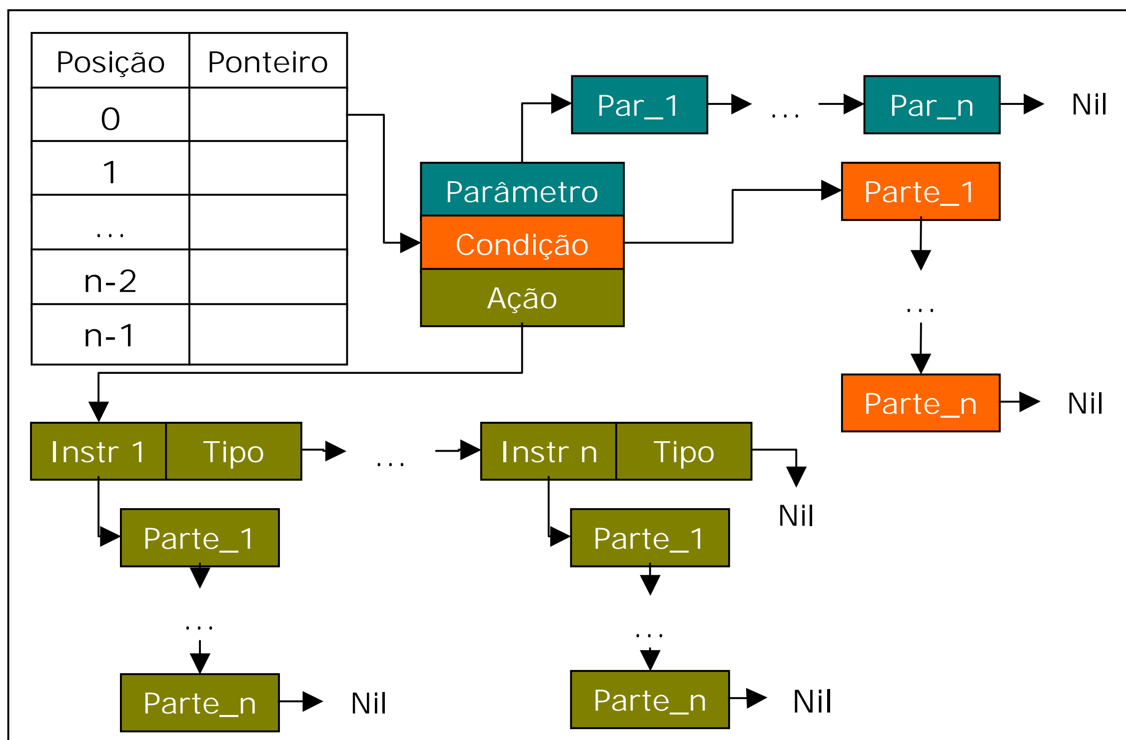

Figura 20 - Esquema da Tabela de Regras

Através desta figura, é possível notar que a estrutura para armazenamento de uma regra qualquer é composta por três campos, a saber:

- *Parâmetro*: consta de um ponteiro para uma lista de condições de disparo. Esta lista compõe-se de nodos, os quais contém apontadores para os sinais que os parâmetros representam. Convém salientar que a lista de parâmetros pode ser vazia, o que significa que a ação será realizada sempre que a regra for solicitada.

- *Condição*: consta de um ponteiro para uma condição a ser verificada. Esta condição compõe-se de uma lista de operandos e de operadores aritméticos, relacionais e lógicos, que serão verificados em tempo de execução e deverão resultar em um valor booleano (*true* ou *false*). Novamente, deve-se notar que este campo pode ser vazio e que somente deixará de sê-lo se a regra for condicional, ou seja, for uma regra do tipo *case* (ou seletora).
- *Ações*: corresponde a um apontador para uma lista de instruções a serem executadas quando os parâmetros e as condições forem satisfeitos. De modo similar às condições, as instruções são compostas por listas de operandos e operadores que serão operados em tempo de execução. São permitidas instruções de atribuição, UP e EMIT. Como os demais campos, este também pode ser vazio.

7.3 O Executor

O executor é considerado o principal módulo dentre os três por ser responsável pela realização das reações indicadas nas tabelas da memória. É através dele que os sinais de entrada são processados, realizando reações e gerando sinais de saída para o ambiente externo.

É possível identificar duas etapas no processo de execução de uma reação: uma etapa de **busca** das instruções e outra de **execução** propriamente dita. Na etapa de busca, o executor recebe o sinal de entrada (e os valores dos parâmetros, caso existam) e procura nas tabelas as ações a serem executadas. Encontradas as ações na memória, passa-se para a etapa de execução, em que as instruções contidas nas ações são executadas (operações aritméticas, atribuições, emissões de sinal, etc).

O algoritmo de execução de uma reação pode ser descrito, em linhas gerais, através dos seguintes passos:

1. Recebimento do sinal de entrada e dos valores dos campos de sinal (se houver);
2. Busca da posição na tabela de transições do autômato correspondente ao sinal de entrada e ao estado corrente (através do vetor indexador);
3. Determinação da validade do sinal (se o sinal não é esperado, ou seja, não existe transição indicada, o mesmo é ignorado);
4. Obtenção das ações a serem realizadas para a linha da tabela encontrada;

5. Busca de ações na tabela de regras e execução das mesmas de maneira seqüencial: executa-se a primeira ação indicada, depois a segunda, e assim sucessivamente, até que todas sejam realizadas. Para cada ação, é verificado se os parâmetros e as condições são verdadeiros;
6. Obtenção do novo estado para o sistema a partir da reação realizada;
7. Devolução do controle para a interface.

8 Implementação de uma Interface Genérica

Para cada aplicação de simulação de sistemas reativos pode ser definida uma interface específica que represente as interações entre o núcleo reativo e o ambiente sendo controlado pelo sistema. Entretanto, abstraindo as particularidades de cada sistema, é possível definir uma interface genérica que simule as interações entre o ambiente e o núcleo reativo, para qualquer sistema simulado.

Este capítulo apresenta o desenvolvimento de uma interface genérica para o novo ambiente de execução da linguagem RS, conforme modelado na seção 6.4. Ao seu final, é apresentado o modelo de comunicação entre executor e esta interface, também referido como Protocolo de Comunicação, que poderá ser utilizado na modelagem de interfaces específicas, como a interface de um relógio digital, conforme exemplificado por Toscani [TOS 93].

8.1 A Interface Visual Genérica

Na seção 6.4, foram estabelecidas diretrizes para a construção das interfaces visuais do sistema, sendo delineadas duas interfaces básicas: a interface de carga e a interface de execução.

A partir destas diretrizes, foram definidas para a interface visual do ambiente cinco janelas básicas:

- **janela principal**: é a janela inicial do ambiente. A partir dela, o usuário tem acesso a um menu fixo, que permite o acesso a todas as outras janelas do programa, se elas estiverem disponíveis no momento. Essa janela possibilita a realização da carga de um programa para a memória e a visualização do código objeto contido no arquivo sendo carregado;
- **janela de configuração**: permite configurar o tamanho das tabelas de transição do autômato e de regras de transição;
- **janela de visualização de tabelas**: mostra, após a carga de um programa, o conteúdo das tabelas. Esta janela só fica disponível após o processo de carga ser realizado com sucesso a partir da janela principal;

- **janela de visualização de erros**: apresenta ao usuário os erros encontrados pelo carregador no processo de carga-tradução do código gerado pelo compilador RS (semelhantemente à janela de tabelas, esta somente se torna disponível se algum erro efetivamente ocorrer);
- **janela de execução**: habilita o usuário a interagir com o programa carregado, sendo parte do seu leiaute montado através de informações coletadas a partir do próprio programa.

É a *janela de execução* que permite simular um programa RS através de estímulos enviados ao executor. Esta janela pode ser configurada conforme o modo de execução escolhido pelo usuário (modo interativo ou programado).

O modo interativo de execução possui a mesma funcionalidade encontrada no ambiente de execução original, ou seja, permite a entrada individual de estímulos quando o usuário desejar. Isso é feito a partir de botões existentes para cada um dos sinais de entrada disponibilizados no programa. Para indicar um sinal de entrada, basta ao usuário clicar no botão desejado. Se o sinal for valorado, será aberta uma janela semelhante à Figura 21 para inserção de valores para os campos de sinal.

Figura 21 - Janela para inserção de valor em campo de sinal

Já o modo programado de execução possui um estilo diferente de simulação. Neste modo, o usuário disponibiliza, através de uma interface apropriada para tal, todos sinais de entrada e os instantes de tempo para cada um dos sinais, como pode ser visto na Figura 28. Isso permite uma simulação com menor interação, em tempo de execução, do usuário com o sistema, similarmente ao que ocorre em sistemas reais.

Na implementação desta interface genérica, o modo programado permite a escolha de uma entre três categorias possíveis para simulação: Temporizado, Não-Temporizado e Passo-a-Passo (Depuração). No modo programado temporizado,

após o início da execução, cada sinal é repassado ao executor, pela interface, no instante de tempo programado previamente. Dessa forma, tem-se uma forma de execução semelhante ao modo interativo, porém com a diferença de ser totalmente pré-programado, não existindo interferência do usuário em tempo de execução.

Já no modo programado não-temporizado, o fator tempo é ignorado e todos os sinais de entrada, previamente indicados na interface pelo usuário, são enviados ao executor em seqüência, um após o outro.

No modo programado passo a passo, o usuário controla o tempo em que o próximo sinal previsto na interface deve ser enviado ao executor. Isto permite que o usuário monitore o comportamento do programa sendo simulado, através da visualização do conteúdo de seus sinais e variáveis.

A *janela principal* interage com o carregador, solicitando que o arquivo escolhido pelo usuário seja colocado na memória do executor. Ao ser escolhido um determinado arquivo, seu conteúdo (autômato e regras na forma produzida pelo compilador RS) é visualizado na própria janela, como ilustra a Figura 22.

Durante a carga do código, o carregador monta uma lista dos erros encontrados. Existindo erros, sua visualização é feita em uma janela específica, a *janela de visualização de erros*. Ela é responsável por tomar a lista de erros elaborada pelo carregador e dispô-los num componente adequado para tal fim. Em geral, os erros indicados são referentes à malformação sintática do código objeto RS. Ao serem descobertos erros em tempo de carga-tradução, é apresentada ao usuário uma mensagem semelhante à ilustrada pela Figura 23. A partir desta caixa de diálogo, o usuário pode requisitar a visualização dos erros, como mostrado na Figura 24.

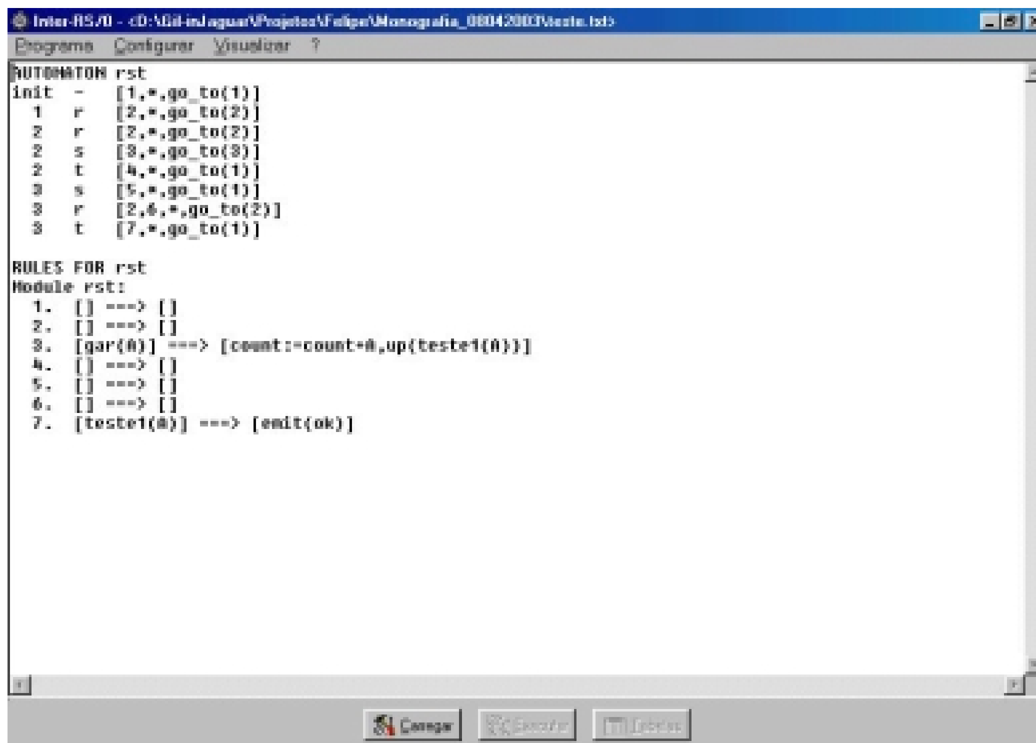

Figura 22 - Janela Principal após seleção de programa

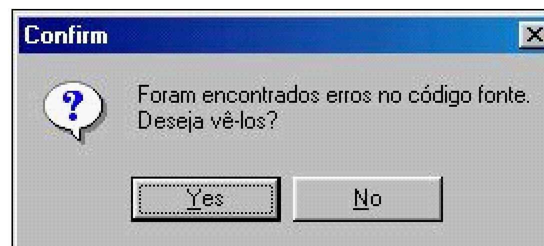

Figura 23 - Indicação de erro de carga

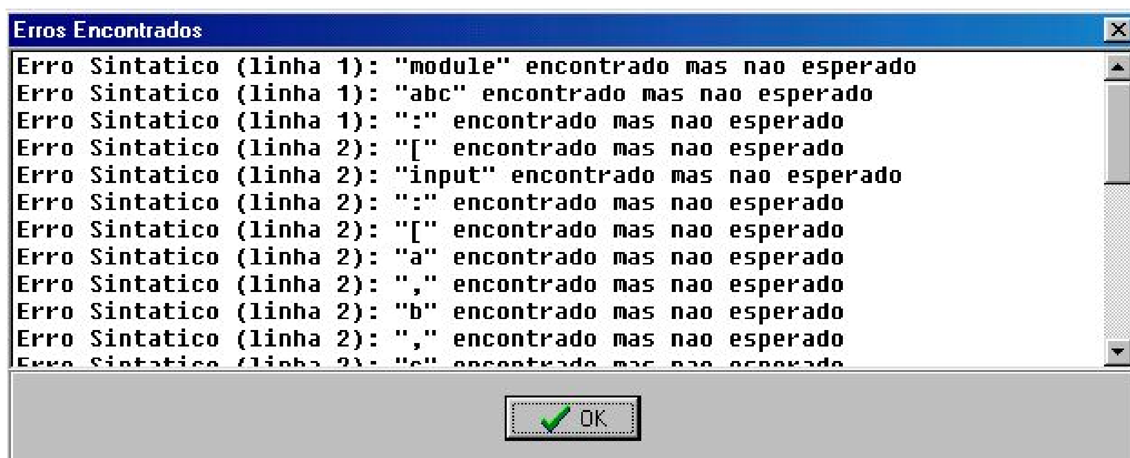

Figura 24 - Janela de Visualização de Erros

Após a visualização dos erros, é possível revê-los através do uso do menu apresentado na janela principal. A Figura 25 ilustra essa facilidade. A mesma figura mostra a possibilidade do uso de teclas de atalho para acelerar e facilitar a interação usuário-sistema, também disponível para as outras opções do menu.

Figura 25 - Menu para visualização da Lista de Erros

Adicionalmente, além de estarem nos menus, alguns dos principais recursos utilizados pelo usuário estão dispostos em botões na parte inferior da janela, agilizando a interação com o ambiente.

No processo de carga, as tabelas de sinais, de variáveis, de transições do autômato e de regras vão sendo montadas para uso do executor. O conteúdo destas tabelas pode ser obtido pelo usuário através da janela de visualização de tabelas (Figura 26).

Visualizar Tabelas

Sinais de Entrada | Sinais de Saída | Sinais Internos | Tabela de Variáveis | Tabela de Simbolos | Automato | Regras

Num.	Estado	Sinal	Acao	Go_to
000	init	-	1,0,0,0,	1
001	1	l	2,0,0,0,	#510
002	1	gar	3,0,0,0,	#508
003	1	s	6,0,0,0,	1
004				
005				
006				
007				
008				
009				
010				
011				
012				
013				
014				
015				
016				
017				
018				
019				
020				

OK

Figura 26 - Visualização das Tabelas

Dentre todas as melhorias apresentadas pelo novo ambiente de execução, as mais amplamente perceptíveis foram implementadas na *janela de execução*. É através dela que o usuário tem a possibilidade de executar simulações com o código carregado, vendo os resultados obtidos e as modificações que o sistema sofre durante a execução.

A janela de execução (Figura 27) é composta por uma *parte estática*, ou seja, determinada em tempo de projeto, e outra construída de maneira *dinâmica*, ou seja, em tempo de execução do ambiente, conforme o programa carregado para execução. A parte estática é constituída da barra de botões, da régua de tempo e das tabelas (excetuando-se o seu conteúdo). A parte dinâmica apresenta as barras de sinais e os botões relativos a cada uma das barras.

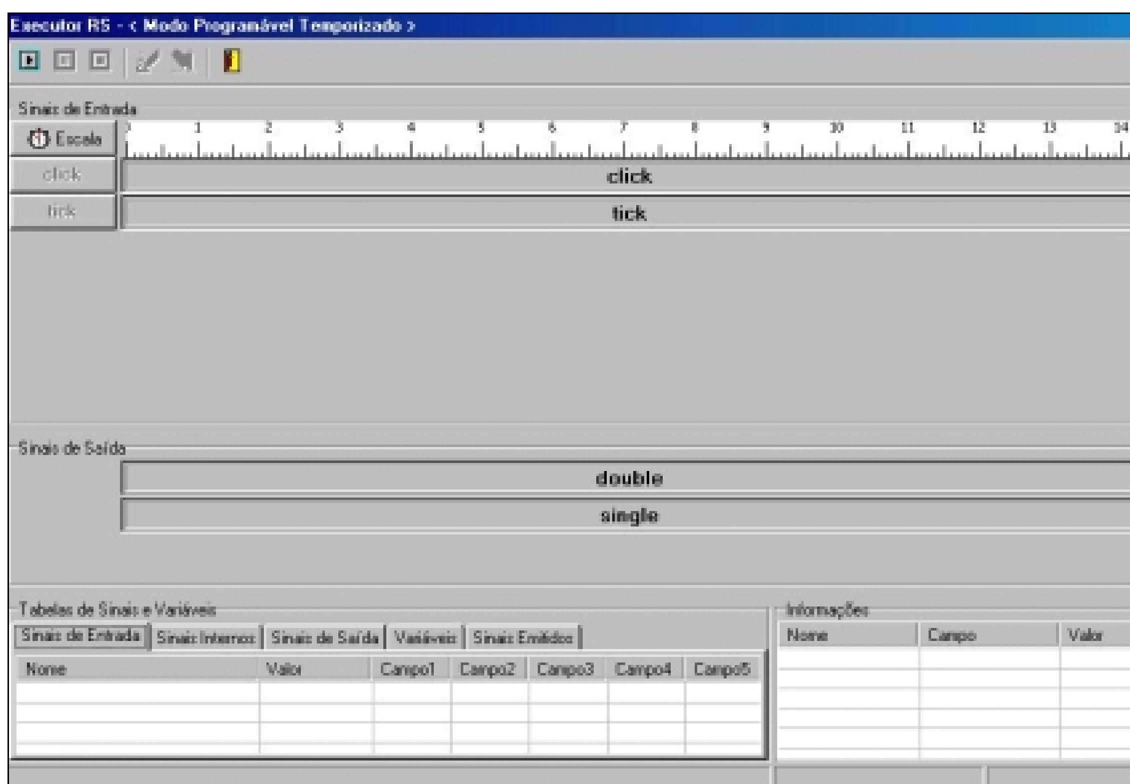

Figura 27 - Janela de Execução

A régua de tempo, que pode ser vista acima, é configurável. A partir dela é possível indicar qual será o período total de simulação para fins de cálculo do momento em que cada um dos sinais será disparado. Para configurá-la, basta que o usuário clique no botão Escala ou na própria régua.

Abaixo da régua de tempo, são dispostas as *linhas de tempo* para marcação de sinais. Elas estão dispostas em dois grupos: o primeiro para os sinais de entrada e o segundo para os sinais de saída. As linhas de entrada são capazes de receber seqüências de sinais que alimentarão o executor num processo de simulação. Basta clicar com o

mouse na barra e posição desejadas para que um sinal seja automaticamente programado na posição. Este comportamento é observável para o modo de execução *programado*. No modo *interativo*, os sinais são indicados, um a um, através de cliques nos botões correspondentes aos sinais de entrada. Se o sinal que está sendo indicado possui campos de valor, automaticamente é aberta uma janela que permite a inserção dos valores para cada um dos campos. As linhas de saída, por outro lado, não são capazes de receber sinais através de intervenção direta do usuário. Elas mostram o resultado das reações em termos de emissão de sinais, nos tempos ocorridos.

Cada um dos sinais inseridos nas linhas de entrada é amplamente configurável. É possível arrastá-los para outras posições dentro da linha em que foram inseridos, modificando a sua posição relativa aos demais sinais. Clicando em um sinal com o botão direito do mouse, é possível acessar um menu de contexto que permite retirá-lo ou até mesmo realizar uma espécie de 'ajuste fino' relativo à posição em que se queira colocá-lo, de maneira semelhante ao arrasto (vide Figura 28). Da mesma forma, apenas clicando sobre a barra de sinal desejado, pode-se observar sua posição relativa de tempo e sua identificação, através da **barra de status**.

Figura 28 - Menu de contexto para sinais

O funcionamento do novo ambiente como um todo será explicado no Anexo 1.

8.2 O Modelo de Comunicação Executor – Interface

Na tentativa de generalizar o processo de criação de interfaces, de maneira a possibilitar o uso do executor com outras desenhadas pelo usuário para uma aplicação mais específica, foi elaborado um modelo para comunicação entre o executor e a interface que fará uso deste.

Este modelo, ou protocolo de comunicação, disponibiliza alguns métodos (ou primitivas) necessários para estabelecer a comunicação entre a interface e o executor. A partir dessas primitivas o executor troca informações com a interface. Tais métodos e propriedades são:

- *create*: cria uma instância do ambiente de execução;
- *tamTSinais*: retém o tamanho da tabela de sinais (número de sinais);
- *tamTVars*: indica o tamanho da tabela de variáveis (número de variáveis);
- *tamTSimb*: armazena o tamanho da tabela de símbolos;
- *tamTAut*: mostra o tamanho alocado da tabela de transições do autômato;
- *tamTReg*: indica o tamanho alocado para a tabela de regras;
- *estadoAtual*: devolve o estado atual do programa sendo executado;
- *sinalEmitido*: contém uma lista de nomes dos sinais de saída emitidos em uma reação;
- *criaExecutor*: instancia o objeto executor que processa os sinais de entrada;
- *destroiExecutor*: destrói o objeto executor previamente instanciado;
- *limpaMemoria*: prepara a memória da máquina virtual para receber um novo programa;
- *zeraDados:* desliga todos os sinais e zera o conteúdo de seus campos e das variáveis do programa;
- *carrega*: chama o carregador para realizar a carga de um programa;
- *pegaLinhaAut*: obtém uma linha da tabela de transições do autômato;
- *pegaLinhaReg*: devolve uma linha da tabela de regras;
- *pegaVariavel*: busca uma variável da tabela de variáveis;
- *pegaSinal*: retorna um sinal da tabela de sinais;
- *pegaSimbolo*: obtém um símbolo da tabela de símbolos;
- *executa*: execução de uma reação dado um sinal de entrada;
- *erros*: contém uma lista dos erros encontrados na carga de um programa.

Os métodos e propriedades que compõem este protocolo são implementados no módulo executor.

9 Conclusões

Este trabalho abrangeu a modelagem e a implementação de um novo ambiente de execução para a linguagem RS, que é independente da linguagem Prolog e já está em funcionamento. O novo ambiente utiliza o código objeto gerado pelo compilador RS, obtendo, a partir dele, as informações necessárias para a execução de um programa RS de forma adequada ao ambiente.

O novo ambiente de execução diferencia-se bastante do original em termos de facilidade de uso. Uma nova forma de interação entre o usuário e o ambiente, baseado em componentes visuais, tais como janelas, botões e menus, foi criada, simplificando a tarefa de utilização do sistema para a execução de programas RS. O uso de interfaces desse tipo praticamente elimina a necessidade de memorização dos comandos e dos passos a serem seguidos para uma simulação.

Para permitir a perfeita utilização do novo ambiente, houve a necessidade de ser efetuada uma modificação no ambiente de RS para proporcionar o salvamento em arquivo do código objeto RS.

Uma das grandes alterações processadas em relação ao modelo original foi a inclusão de novos modos de execução de um programa, com destaque ao modo *temporizado*. Através deste modo de execução, um programa pode ser executado de forma automática através apenas da indicação dos sinais desejados e dos instantes de tempo para estes na interface elaborada. Assim, os sinais são processados no instante pré-determinado pelo usuário. Também é possível uma execução semelhante a esta, chamada de *não temporizada*, que se diferencia da anterior por desconsiderar o tempo entre um sinal e o consecutivo, além, é claro, do modo de execução já utilizado no ambiente original.

Apesar de não fazer parte do escopo definido para este trabalho, também foi implementado um modo de depuração bastante simples, que permite a verificação do comportamento de um programa RS através da verificação dos valores de seus sinais e variáveis, os quais ficam dispostos diretamente na interface.

Também foi também elaborado um protocolo de comunicação entre a interface genérica construída para este novo ambiente e a máquina virtual implementada que

executa o código compilado RS. A definição deste protocolo permite, efetivamente, a elaboração de novas interfaces que sejam mais específicas para a aplicação a que se destinam, proporcionando uma maior flexibilidade no uso da máquina virtual implementada.

Para o projeto e a implementação deste trabalho, foi necessário o uso de conhecimentos previamente adquiridos em diversas áreas da computação, tais como: algoritmos, estrutura de dados, programação em lógica, programação orientada a objetos, linguagens formais, compiladores, interface humano-computador, dentre outros.

Além desses temas, o estudo das características dos sistemas reativos, de suas aplicações, tal como em sistemas embarcados e de tempo real, e da linguagem reativa RS proporcionou um complemento aos conhecimentos já adquiridos. Da mesma forma, a reunião de esforços em torno do estudo desses temas e da implementação deste novo ambiente propiciaram a oportunidade de fornecer alguma informação que possibilite a continuação futura deste trabalho.

Devido à natureza da estrutura do código objeto apresentado, o presente trabalho também proporcionou a oportunidade de trabalhar com estruturas de dados complexas para o armazenamento das informações obtidas a partir desse código.

Atualmente, o novo ambiente de execução da linguagem RS trabalha somente com códigos-objeto na sintaxe de RS-0. Códigos compostos por mais de um módulo ou que possuam construções de linguagem mais sofisticadas, tais como variáveis do tipo *record*, sensores e regras de exceção, ainda não são tratados. Seria interessante que códigos com essas características também pudessem ser simulados através de um ambiente desse porte.

Outro aspecto a ser trabalhado futuramente é a otimização das estruturas de dados utilizadas no armazenamento de um programa RS, tanto em espaço ocupado quanto em tempo de acesso aos dados nele contidos, visto que não houve preocupação especial neste sentido em termos de projeto.

Também, constatou-se que o processo de carga-tradução, feito num só passo, poderia ter sido simplificado se o código objeto fosse apresentado de uma maneira diferente, que privilegiasse as estruturas de dados contidas no programa. O mesmo, a partir de algumas modificações no código de RS, poderia conter, de forma agregada, as estruturas de dados contidas no programa, facilitando o processo.

ANEXO 1 – Manual de Utilização

O uso do novo ambiente desenvolvido para a linguagem RS pode ser considerado bastante simples e intuitivo. Para facilitar a operação, principalmente por usuários iniciantes na área de simulação de sistemas reativos, nas seções seguintes serão mostrados alguns passos para a utilização deste ambiente de execução. Para tanto, o sistema composto pelo ambiente original (compilador RS) e pelo novo ambiente de execução deve estar instalado em um sistema computacional operando sob a plataforma Windows.

Deve ser observado que o presente manual não é autocontido, ou seja, em alguns momentos são feitas referências a determinadas partes (sobretudo figuras) contidas no demais capítulos da monografia.

A. Ativação do Novo Ambiente de Execução

O correto funcionamento do novo ambiente de execução passa, necessariamente, pela obtenção do código necessário. Esse código é fundamental para todo o processo de execução neste novo ambiente. Para poder fazer uso dele, o usuário deverá ter em mãos um código objeto RS.

Para que esse código possa ser obtido, o usuário deverá ativar o ambiente Prolog e carregar nele o ambiente de execução RS original (Figura 13). Terminada essa etapa, realiza-se a compilação do programa-fonte RS que se deseja simular, de acordo com o que foi explicado na seção 4.1.

A partir da obtenção do código objeto RS, é possível agora ativar o novo ambiente de execução para realizar a simulação deste programa. Isso é feito através de um duplo-clique no arquivo executável correspondente ao programa, o qual apresentará uma janela semelhante à apresentada na Figura 22.

B. Escolha do Programa

A definição do arquivo de programa a ser carregado para execução é iniciada com a escolha do item de menu Arquivo|Abrir (ou através da tecla de atalho CTRL+A),

que abre de uma janela do tipo “Abrir Arquivo”, muito comum em programas baseados nesse estilo de interação (Figura 29). A partir dessa janela, o usuário tem condições de navegar pela estrutura de diretórios do sistema em busca do arquivo desejado.

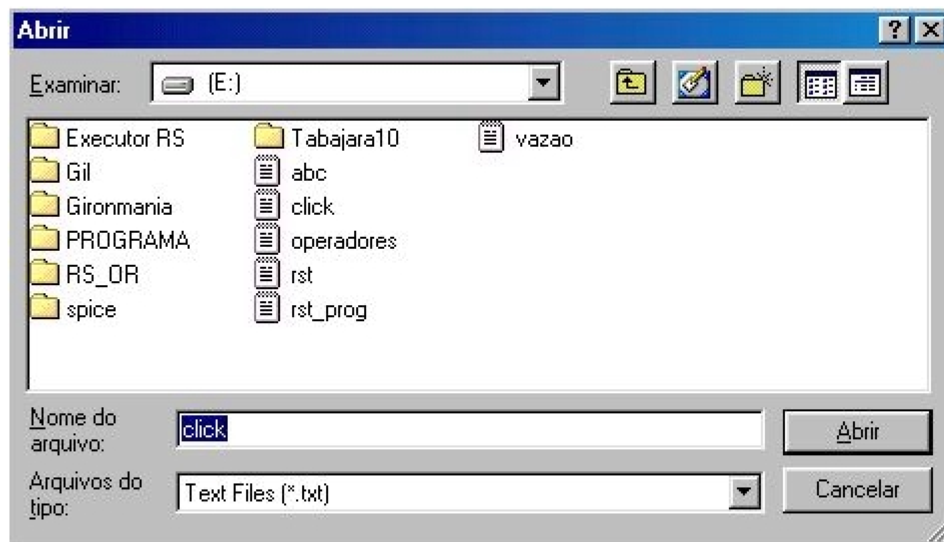

Figura 29 - Janela de abertura de arquivo

Uma vez escolhido o programa, seu conteúdo é apresentado na janela principal, conforme mostrado na Figura 22

C. Configuração do Ambiente da Máquina Virtual

Antes de realizar a carga do programa para posterior simulação, o tamanho das tabelas que armazenarão seu conteúdo deve ser configurado. Isso é feito através da utilização da *janela de configuração*, cujo formato pode ser visualizado através da Figura 30.

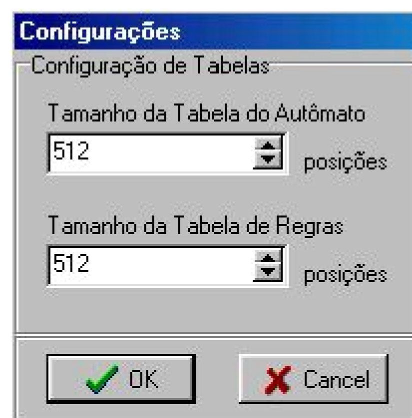

Figura 30 - Janela de Configuração

Através dela, é possível determinar o tamanho das tabelas de regras e de transições do autômato para um dado programa. Inicialmente, ambas possuem um

tamanho mínimo fixo de 512 posições disponíveis para a inserção de dados. Esse valor é suficiente para a maioria dos programas, visto que, em geral, os mesmos contam com poucas linhas de código.

D. Carga de Programa

Escolhido o arquivo, o carregador pode ser acionado. Isso é feito através do uso do botão 'Carregar', disposto na parte inferior da janela principal (Figura 31). Quando isso ocorre, todo o processo de reconhecimento e carga do código é realizado, sendo reportados os erros ocorridos nesta etapa, se houver algum.

Após a carga ser concluída com sucesso, a execução do programa fica habilitada e é possível ver o conteúdo das tabelas que compõem a memória da máquina virtual, através do botão 'Tabelas', disposto junto ao anterior.

Figura 31 - Botões disponíveis

E. Execução do programa

Terminado o processo de carga do código objeto para a memória da máquina virtual, a execução do programa pode ser ativada a qualquer momento, através do uso do botão 'Executar' (parte inferior da janela). Este botão corresponde à escolha do item de menu Arquivo|Executar ou da combinação de teclas CTRL+E (Figura 32).

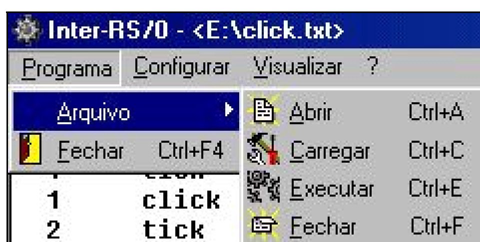

Figura 32 - Menu Arquivo

O uso deste botão apresenta ao usuário uma janela semelhante àquela indicada na Figura 33. Através dela, o usuário deverá decidir entre um dos modos de execução disponíveis: modo interativo, modo programado temporizado, modo programado não-temporizado e modo programado passo-a-passo. Cada um desses modos determina uma maneira diferente de executar o código carregado, como foi explicado na seção 8.1.

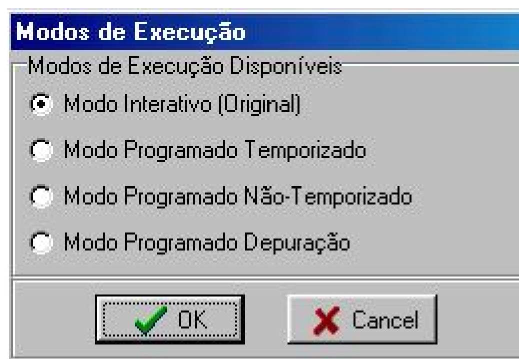

Figura 33 - Modos de Execução

Se o modo de execução escolhido for o modo programado temporizado, uma janela de *configuração de escala*, como a indicada pela Figura 34, será apresentada ao usuário. Através dela, deverá ser indicado o *tamanho do intervalo* de tempo total em que a simulação ocorrerá, considerando a unidade de medida escolhida (que pode variar de segundos até horas). Além disso, um *intervalo de escala* também deverá ser indicado para definir o intervalo entre números mostrados na régua.

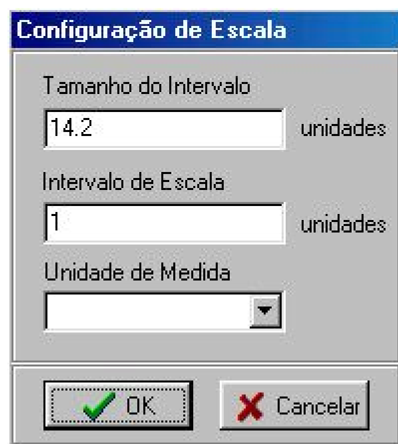

Figura 34 – Configuração de Escala

Superada essa etapa, o usuário entrará em contato com a janela de execução, que é a parte da interface dedicada ao processo de simulação de programas. Seu formato pode ser visto através da Figura 22.

O comportamento da *janela de execução* durante a execução de um programa RS, tanto nos modos programados quanto no modo interativo, é muito semelhante. Para iniciar a execução do programa, utiliza-se o botão *Rodar*, colocado na parte superior esquerda da janela, conforme visto na Figura 22. A ativação deste botão determina a realização das ações de inicialização do programa e preparação do núcleo para recepção de estímulos (sinais) externos.

Figura 35 – Botões de execução

Além do botão ‘Rodar’, que é o primeiro da esquerda para a direita na figura, existem, ainda, outros cinco botões colocados na parte superior da janela, que são, respectivamente: o botão ‘Pause’, que interrompe a execução de um programa sendo simulado em um dos modos programados; o botão ‘Parar’, que cancela a execução do programa; o botão ‘Limpar sinais de entrada’, que limpa os sinais de entrada dispostos na interface; o botão ‘Limpar sinais de saída’, que limpa os sinais de saída oriundos de uma execução de programa anterior; e o botão ‘Sair’, que fecha a janela. O botão Pause assume grande importância no modo de execução passo-a-passo (depuração). Nesse modo, todos os sinais de entrada são dispostos nas linhas de tempo e a indicação do instante em que o próximo sinal será enviado ao executor é feito através deste botão, como pode ser visto através da Figura 35.

Uma das poucas diferenças visíveis entre os modos de execução situa-se na maneira como os sinais são enviados ao executor. No modo interativo, cada sinal é passado ao executor pelo usuário através do uso de botões dispostos no lado esquerdo da janela, correspondendo um botão para cada sinal de entrada do programa. Um clique em botão indica a entrada de sinal nesse modo de interação (Figura 36). Se, porventura, um sinal for valorado, ao clicar no botão correspondente ao mesmo, será aberta uma pequena janela para inserção de valores nos campos de sinal.

Nos demais modos, os *sinais de entrada* são dispostos pelo usuário nas *linhas de tempo* existentes para cada um dos sinais de entrada. A partir dessas linhas, é possível modificar a posição relativa do sinal de entrada colocado, através do arrasto do mesmo para uma outra posição desejada. Também é possível a utilização do botão direito do mouse para obter informações e acessar funcionalidades relativas a cada um dos sinais dispostos através de um menu de contexto, conforme visto através da Figura 28. Da mesma forma que no modo programado, ao ser inserido um sinal valorado, será aberta uma janela para inserção de valores nos campos.

Figura 36 - Botões para entrada de sinal no modo Interativo

As cores indicadas nos sinais de entrada têm seu significado. Quando um sinal é colocado na linha, sua cor corresponde a um tom em azul-água. Quando um sinal já colocado é clicado, sua cor muda para vermelho. Após a execução do programa, sinais de entrada na cor azul-forte indicam execução correta, enquanto que sinais de cor preta indicam erro na utilização daquele sinal pelo sistema. Algumas das cores citadas podem ser vistas através da Figura 28.

Abaixo das linhas de tempo para os sinais de entrada, existem as linhas correspondentes aos *sinais de saída*. Através delas, são indicados os sinais emitidos pelo programa nos instantes em que os mesmos ocorrerem.

Na parte inferior da janela, estão dispostas algumas tabelas que ajudarão o usuário a perceber o comportamento do programa que está sendo executado, sobretudo quando é utilizado o modo de execução passo-a-passo. Através dessas tabelas, ele terá condições de perceber o conteúdo de variáveis, a situação de sinais do sistema, o estado atual do programa, e outras informações úteis.

Bibliografia

[AHO 88] AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D.. **Compilers: Principles, Techniques and Tools**. Addison-Wesley, 1988.

[ARN 98] ARNOLD, Gustavo V.. **Uma Extensão da Linguagem RS para o Desenvolvimento de Sistemas Reativos Baseados na Arquitetura de Subsunção**. Porto Alegre: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1998.

[CAS 94] CASPI, Paul; GIRAULT, Alain; PILAUD, Daniel. **Distribuited reactive systems**. In: International Conference on Parallel and Distribuited Computing Systems, 1994, Las Vegas, USA. Proceedings... [S.l: s.n.], 1994. p.101-107. Disponível em <http://www.inrialpes.fr/bip/people/girault/Publications/Pdcs94/>. Acesso em: 22 jan. 2003.

[GIO 98] GIORGI, Ulisses P.. **A distribuição da linguagem RS**. Porto Alegre: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1998.

[HAL 93] HALBWACKS, Nicolas. **Sinchronous Programming of Reactive Systems**. Dodrecht: Kluwer Academic Publishers, 1993.

[HAR 85] HAREL, David; PNUELI, Amir. **On the Development of Reactive Systems**. Logics And Models Of Concurrent Systems, NATO ASI Series, Vol F13, p.477-498, Springer-Verlag Berlin Heidelberg, 1985.

[LIB 2001] LIBRELOTTO, Giovani R.. **Um Compilador para a linguagem RS distribuída**. Porto Alegre: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2001.

[MAT 99] MATTOS, Julio C.B. de. **Proposta de Geração de Código VHDL a partir da Linguagem RS**. Porto Alegre: UFRGS, Trabalho Individual.

Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1999.

[MAT 2000] MATTOS, Julio C.B. de. **Geração de código no projeto de sistemas reativos a partir da linguagem RS**. Porto Alegre: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2000.

[PAL 97] PALAZZO, Luiz A.M.. **Introdução à Programação Prolog**. Pelotas: EDUCAT, 1997.

[PRI 2001] PRICE, Ana M. de A.; TOSCANI, Simão S.. **Implementação de Linguagens de Programação: Compiladores**. 2ed. Porto Alegre: Sagra-Luzzatto, 2001.

[SOU 99] SOUZA, C. S. et al.. **Projeto de Interfaces de Usuário: Perspectivas Cognitiva e Semiótica**. Anais da Jornada de Atualização em Informática, XIX Congresso da Sociedade Brasileira de Computação, Rio de Janeiro, julho de 1999. Disponível em: <http://www.dimap.ufrn.br/~jair/piu/JAI_Apostila.pdf> Acesso em: 22 jan. 2003.

[TOS 93] TOSCANI, Simão S.. **RS: Uma Linguagem para Programação de Núcleos Reactivos**. Lisboa: Universidade Nova de Lisboa, Dissertação de Doutorado. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1993.