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

CONSOLIDAÇÃO DO AMBIENTE EXEC-RS COM A INCLUSÃO DE MECANISMOS DE EXCEÇÃO E INTERFACES DEMONSTRATIVAS

Allan Sampaio Pires

Pelotas, 2006

Allan Sampaio Pires

CONSOLIDAÇÃO DO AMBIENTE EXEC-RS COM A INCLUSÃO DE MECANISMOS DE EXCEÇÃO E INTERFACES DEMONSTRATIVAS

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação do Instituto de Física e Matemática da Universidade Federal de Pelotas, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Gil Carlos Rodrigues Medeiros, MSc.

Pelotas, 2006

<Verso da Folha de Rosto>

<Ficha Catalográfica>

Consolidação do Ambiente Exec-RS
com a Inclusão de Mecanismos de Exceção
e Interfaces Demonstrativas

por

Allan Sampaio Pires

Monografia defendida pelo autor e aprovada em ____ de _____ de 2006
pela banca examinadora composta pelos professores abaixo relacionados.

Banca Examinadora:

Prof. Gil Carlos Rodrigues Medeiros, MSc. .....

Prof. José Luís Almada Güntzel, Dr. .....

Prof. Marcello da Rocha Macarthy, MSc. .....

Agradecimentos

Aos meus pais, Cláudio Vilmar Pires e Maria Eli Sampaio Pires, por sempre me apoiarem em todos os caminhos que escolhi, e por todo incentivo e amor.

Aos meus irmãos Cláudia e Alex e também a minha avó Neli Vergara Sampaio, por sempre estarem perto e de alguma forma ajudando neste trabalho.

Ao meu orientador, Gil Medeiros, pelo apoio e ensinamentos transmitidos neste trabalho, além de ser um amigo que está sempre disposto a ajudar.

Aos colegas que conviveram comigo no período da graduação, na qual se tornaram grandes amigos.

A todos os amigos que estiveram presentes comigo nesta longa caminhada, tanto nos momentos bons como nos momentos difíceis.

Resumo

Sistemas Reativos Síncronos são sistemas que continuamente devem responder a estímulos externos numa ordem desconhecida. Para a programação desses sistemas, existem linguagens específicas, entre elas a Linguagem RS (Reativa Síncrona), cujo ambiente de execução foi originalmente desenvolvido na linguagem Prolog. Posteriormente, foi desenvolvido, com a linguagem Object Pascal, o ambiente de execução Exec-RS, como uma alternativa mais amigável para a execução destes programas. Este trabalho, implementa melhorias que podem ser acrescidas ao ambiente Exec-RS, tais como o mecanismo de tratamento de exceções definido originalmente para o simulador do núcleo reativo e a criação de interfaces de execução específicas para programas típicos existentes, visando explorar as características do protocolo de interação definido no ambiente Exec-RS. Anteriormente, a camada de interface dos sistemas reativos simulados era representada por uma única interface genérica de execução, que também fora implementada com base neste protocolo, mas sem o tratamento visual das particularidades de cada aplicação. O resultado deste trabalho promove o ambiente Exec-RS ao nível de total compatibilidade com a versão plena da linguagem RS e permite demonstrar de forma mais amigável a forma como os diversos sinais interagem com o sistema.

Lista de ilustrações

Lista de Tabelas

Sumário

1 Introdução

Existem sistemas que devem responder, de forma imediata, a estímulos externos aleatórios. Esses sistemas, que tem um relacionamento dinâmico com o ambiente, são denominados reativos (OLIVEIRA, 2005).

Este tipo de sistemas é utilizado em diversos tipos de aplicações e podem ser citados exemplos na área automobilística, como freios ABS, máquinas de venda automática, usinas nucleares e video-games. Por isso, os sistemas reativos devem ser confiáveis e seguros, pois possíveis erros que ocorram, poderão causar sérios danos econômicos e até mesmo pôr em risco vidas humanas (OLIVEIRA, 2005).

Geralmente, esses sistemas são divididos em três camadas: a camada do núcleo reativo, onde são tratados os sinais que chegam do exterior; uma camada de interface, que relaciona o ambiente externo com o núcleo reativo; e a camada de manipulação de dados, que realiza operações triviais exigidas pelo núcleo reativo e, geralmente, é implementada por procedimentos externos (ZSCHORNACK, 2003).

Para esses sistemas, existem diversas linguagens de programação, sendo que uma delas é a linguagem RS, cujo nome significa linguagem Reativa Síncrona. RS foi desenvolvida por Toscani (1993) e sua concepção baseia-se no uso de regras do tipo "`sinal & condição ==> reação`", que declara as ações a serem tomadas com o recebimento de sinais (OLIVEIRA, 2005). Esta linguagem possui quatro versões, que são: RS-0, RS-1, RS-2, RS-Plena. É possível fazer os mesmos programas nas quatro versões da linguagem. Entretanto, existem facilidades maiores para programação à medida que se avança nas versões da linguagem.

Originalmente, a linguagem RS foi implementada integralmente em Prolog, na forma de um compilador junto com um ambiente de execução (ZSCHORNACK, 2003). Em função disso, a interação com as aplicações foi projetada através de linhas de comando, tornando o ambiente pouco amigável para os usuários.

Motivado pelo interesse de tornar o ambiente mais amigável, Zschornack (2003) propôs e desenvolveu o Ambiente Exec-RS, utilizando o ambiente Delphi (CANTÚ, 2000) para programação. O projeto do novo ambiente foi baseado no conceito de WIMP (Windows, Icons, Menus and Pointers), o que o torna muito mais amigável para o usuário. Inicialmente o ambiente Exec-RS foi desenvolvido para suportar códigos objeto da linguagem RS-0 (RS versão 0), que possui um conjunto básico de recursos. Posteriormente, o ambiente foi melhorado por Oliveira (2005), que implementou os recursos para possibilitar a utilização das versões RS-1 e RS-2 da Linguagem, deixando em aberto uma possível ampliação do ambiente, para suportar a versão RS-Plena. Para consolidação do ambiente, seria necessária a incorporação do mecanismo de tratamento de exceções, de forma que pudesse receber programas baseados em todos os recursos da linguagem RS-Plena. O tratamento de exceções, no contexto da linguagem RS, torna-a especialmente poderosa, pois gera mudanças bruscas em caso de condições especiais que ocorram no programa (TOSCANI, 1993).

O ambiente de execução Exec-RS possui os seguintes componentes: interface genérica, memória, carregador, executor e protocolo de comunicação entre a camada de interface e o executor (protocolo de interação).

O **carregador** recebe um código objeto RS, gerado pelo compilador, e faz um processo de conversão para carregá-lo de forma estruturada na **memória** emulada no ambiente de execução. O **executor** se encarrega de buscar e executar as instruções armazenadas nesta memória, de acordo com os sinais que chegam pela **interface de usuário**, através do **protocolo de comunicação** definido.

Outra característica importante do ambiente de execução Exec-RS é sua interface genérica, que, independente do programa executado, terá sempre as mesmas características gráficas. Entretanto, alguns sistemas reativos exigem, para melhor compreensão do usuário, interfaces específicas. Como exemplos, podem ser citados o relógio digital e o jogo do reflexo (TOSCANI, 1993), que possuem características próprias e, por serem programas de complexidade mais avançada, exigem uma forma mais clara de interação com o usuário. Tendo em vista que o núcleo reativo e a interface genérica possuem um protocolo de comunicação que os interliga, é possível a construção de interfaces específicas sem a necessidade de alteração do núcleo reativo.

Com isso, o objetivo desse trabalho pode ser descrito nos seguintes itens:

- implementar os mecanismos de tratamento de exceções, consolidando o ambiente Exec-RS, com o propósito de aceitar a linguagem RS-Plena;

- implementar interfaces específicas para exemplos típicos de programas da linguagem RS;

- fazer os ajustes necessários na versão anterior do Exec-RS para adequá-lo à incorporação dos novos recursos.

Após esta introdução, o presente trabalho se desenvolve em mais sete capítulos. O segundo capítulo fala sobre os sistemas reativos e a linguagem RS. O ambiente Exec-RS e sua interface gráfica são abordados no terceiro capítulo. No quarto capítulo, é abordada apenas a linguagem RS-plena, para destacar o mecanismo de tratamento de exceções. O quinto capítulo é dedicado à análise de requisitos e à implementação para consolidação do ambiente Exec-RS. As implementações da interface são apresentadas no sexto capítulo. Finalmente, no sétimo capítulo apresentam-se as conclusões e considerações finais.

2 Sistemas Reativos e Linguagem RS

Este capítulo abordará aspectos introdutórios de sistemas reativos e apresentará, também, a linguagem RS, que é uma linguagem para programação desse tipo de sistemas.

2.1 Sistemas Reativos

Para a classificação de sistemas computacionais, existem diversas dicotomias. Uma delas, proposta por Harel (1985), visa separar sistemas facilmente tratáveis de sistemas problemáticos, a partir da qual foram propostas duas categorias de sistemas computacionais: sistemas transformacionais e sistemas reativos (TOSCANI, 1993).

Sistemas transformacionais são os sistemas que, a partir de um conjunto de dados, produzem respostas. Exemplos desses sistemas podem ser observados em programas que contenham funções matemáticas para solução de problemas numéricos.

Sistemas reativos são os sistemas que devem responder continuamente a estímulos externos que chegam em ordem desconhecida (Oliveira, 2005). Esses sistemas, segundo Toscani (1993), “caracterizam-se por interagir fortemente com o ambiente”. São observados exemplos desses sistemas computacionais em diversas aplicações, como em usinas nucleares, o sistema de freios ABS de carros e até mesmo em video-games. Entretanto, esse tipo de sistema não deve ser confundido com sistemas interativos, pois, apesar de também responderem a estímulos externos, o tempo de resposta desses sistemas é determinado pelo próprio sistema, enquanto nos sistemas reativos o tempo de resposta é determinado pelo ambiente.

O comportamento de sistemas reativos pode ser caracterizado por uma seqüência de três passos, que são executados de forma cíclica:

- espera por um sinal externo que estimule o sistema;
- tratamento do sinal externo recebido, gerando respostas do sistema;
- emissão de sinais externos como resposta aos estímulos recebidos.

Deve ser observado que, ao emitir sinais, o sistema está influenciando o ambiente externo.

2.1.1 Organização de Sistemas Reativos

Os sistemas reativos podem ser organizados de forma geral em três camadas:

- camada de interface, que é encarregada de captar os estímulos externos e encaminhá-los ao programa reativo, de forma que possam ser tratados, e enviar as respostas dessas computações ao ambiente;
- camada do núcleo reativo, que é responsável pelo tratamento dos estímulos recebidos pelo programa e, conseqüentemente, produz as respostas esperadas no exterior;
- camada de manipulação de dados, que é responsável por computações triviais requeridas pelo núcleo reativo.

2.1.2 Hipótese de Sincronismo

Para funcionamento correto de programas reativos, é importante considerar que as operações realizadas, presumivelmente, ocorrem em tempo zero, ou seja, as computações ocorrem em passos atômicos discretos, onde o tempo é ignorado (CASPI, 1994 apud ZSCHORNACK, 2003). Como é impossível realizar essas computações em tempo zero, para a hipótese de sincronismo funcionar, basta que o ambiente externo mantenha-se inalterado durante as computações do núcleo reativo. Isso significa que um sinal externo deve ser tratado totalmente antes da chegada de outro sinal externo.

2.2 Linguagem RS

Para a programação de sistemas reativos, existem diversas linguagens, dentre as quais podem ser citadas Esterel (BERRY, 1989 apud TOSCANI, 1993), Statecharts (HAREL, 1987 apud TOSCANI, 1993), Lustre (CASPI, PILAUD, HALBWACHS e PLAICE, 1987 apud TOSCANI, 1993) e Signal (LE GUERNIC, BENVENISTE, BOURNAL e GAUTHIER, 1985 apud TOSCANI, 1993). Uma dessas linguagens foi desenvolvida por Toscani (1993) e nomeada de linguagem RS (Reativa Síncrona). Essa linguagem permite combinar programação concorrente com execução de autômatos determinísticos.

A Linguagem RS destina-se a programação dos núcleos reativos de sistemas reativos. Sintaticamente, a linguagem pode ser expressa através de regras de reação, que são representadas na seguinte forma:

Sinal & Condição ✍ Ação .

Esta expressão tem como significado semântico o seguinte: se ao chegar o sinal, a condição for verdadeira, então a ação será executada.

2.3 Versões Evolutivas da Linguagem RS

A linguagem RS possui quatro versões evolutivas, de tal forma que a versão posterior sempre acrescenta facilidades à versão anterior. Na versão inicial, conhecida com RS-0, são implementados mecanismos básicos da linguagem, que servem como base para todas as versões seguintes. Nas duas versões seguintes, são acrescentadas apenas facilidades, que tornam a programação mais confortável, sendo que todos os mecanismos que as diferem podem ser simulados nas versões anteriores. Entretanto, a última versão - conhecida como versão plena - traz um mecanismo que aumenta o poder da linguagem, não sendo facilmente simulado nas outras versões. Esse mecanismo é conhecido como mecanismo de exceção.

A Tabela 1 ilustra a evolução da linguagem.

Tabela 1 - Versão da linguagem e características.

Versão	Características
RS-0 (Básica)	Apresenta um conjunto mínimo de mecanismos; Programas formados por um único conjunto de regras de reação.
RS-1	Permite: - representar sensores; - utilizar variáveis do tipo “Record”; - chamar procedimentos externos; - emitir um mesmo sinal mais de uma vez numa mesma reação; - ler o valor de qualquer sinal em qualquer regra.
RS-2	Facilita a estruturação dos programas usando caixas de regras e módulos; Possibilita o uso de sinais temporários.
RS-Plena	Inclui mecanismo para tratamento de exceções.

2.3.1 Linguagem RS-0

A linguagem RS-0 é a versão básica, pois ela possui somente o necessário para a programação, não contendo grandes mecanismos que facilitem o trabalho de programação. A Figura 1 mostra a sintaxe de um programa baseado nesta versão da linguagem.

As regras de reação, como já havia sido mencionado anteriormente, possuem a forma **`sinal & condição`** ✍ **`ação`**.

Observe-se que as declarações devem aparecer na ordem em que se encontram na figura, exceto aquelas que possuem listas vazias, que podem ser omitidas. A lista de *input* não poderá ser vazia, pois não faz sentido um programa destinado a responder estímulos externos, que não receba esses estímulos (TOSCANI, 1993).

```
Module M:
[  input: I,
   output: O,
   signal: S,
   var: V,
   Initially: C,
   R
]

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

Figura 1 - Forma do programa reativo na linguagem RS-0.
Fonte: TOSCANI, 1993, p.44

Nesta versão da linguagem, assim com em todas as outras, existem três tipos de sinais, que são:

- Sinais de entrada, que vem do ambiente e estimulam o programa reativo, sendo declarados como *input*;

- Sinais de saída, que são emitidos como respostas dos estímulos para o ambiente externo, sendo declarados como *output*, e

- Sinais internos, que servem para comunicação e sincronização internas.

As variáveis na linguagem RS-0 devem ser simples e declaradas em letras minúsculas, por exemplo:

```
Var: [a, b, c]
```

Os operadores e comandos utilizados em todas as versões da linguagem são baseados em termos de Prolog. Isso se deve ao fato de que o compilador e o ambiente de execução desenvolvidos por Toscani (1993) foram originalmente implementados na linguagem Prolog e, na sua definição, foram absorvidas as características desta linguagem.

2.3.2 Linguagem RS-1

A linguagem RS-1 possui a mesma estrutura da versão anterior, porém utiliza alguns mecanismos a mais. Ao todo são cinco novas incorporações, que podem ser resumidas da seguinte forma (TOSCANI, 1993):

- possibilidade de emissão de um mesmo sinal mais de uma vez numa mesma reação;
- chamada de procedimentos externos;
- representação de sensores, que podem ser lidos em qualquer reação;
- possibilidade de leitura de valores em qualquer regra;
- utilização de variáveis do tipo *record*.

2.3.3 Linguagem RS-2

A linguagem RS-2 inclui recursos que facilitam a estruturação do programa e possibilitam o uso de sinais temporários e permanentes. Ao todo são três novos mecanismos, que podem ser resumidos da seguinte forma:

- caixas de regras, que agrupadas logicamente, ajudam a tornar o programa mais claro;
- sinais internos temporários (*t_signal*) e permanentes (*p_signal*), que servem para comunicação e sincronização internas, e
- módulos, que permitem organizar o programa de forma a constituir partes independentes, que atuarão em paralelo.

2.3.4 Linguagem RS-Plena

A linguagem RS-Plena é a última na escala evolutiva da linguagem RS. Possui um maior poder que as suas versões anteriores porque acrescenta um novo mecanismo, que é conhecido com mecanismo de tratamento de exceções.

Devido a sua importância, essa versão da linguagem será abordada no capítulo 4, onde serão aprofundados os conceitos de mecanismos de exceções.

Após serem abordadas as quatro versões da linguagem, os operadores e comandos da linguagem RS podem ser observados através da Tabela 2 e Tabela 3.

Tabela 2 - Operadores da linguagem RS.

Operador	Descrição
rs_prog	Inicia a definição de um programa RS
module	Declaração de módulo
box	Inicia a definição de uma caixa de regras
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
cf	Especifica uma função de composição de sinal
var	Declaração de variáveis
initially	Comandos a serem executados antes do início do programa
on_exception	Define os eventos de exceção de um programa
===>	Vincula ação a uma condição de disparo em uma regra
--->	Vincula ação a uma condição booleana em regras do tipo
:	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
~, &, v	Operadores Lógicos: Não, E, OU
=, =:=	Igualdade
~=, =\=	Diferença
>=, =>	Maior ou Igual
<=, =<	Menor ou Igual
>, <	Maior, Menor
+ , -, *, /	Soma, Subtração, Multiplicação e Divisão Real
//	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

Fonte: OLIVEIRA, 2005, p.18

Tabela 3 - Comandos da Linguagem RS.

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
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(caixa)	Habilita a caixa de regras especificada

Fonte: OLIVEIRA, 2005, p.19

2.4 Código Objeto da Linguagem RS

O código objeto da linguagem RS é gerado por um compilador desenvolvido por Toscani (1993), que posteriormente sofreu alterações por Zschornack (2003) e Oliveira (2005). Como originalmente o código objeto era gerado e imediatamente executado, em função da forma integrada de atuação do compilador e do ambiente de execução, não existia a necessidade de salvar o código compilado. Com a criação do ambiente Exec-RS, passou a existir a necessidade de obtenção do código objeto em um arquivo. Posteriormente, novas necessidades foram aparecendo, determinando a implementação de alterações no compilador. Como conseqüência, alguns ajustes no código objeto foram necessários para melhor funcionamento do Exec-RS. Na versão do ambiente Exec-RS produzida por Oliveira (2005), foi necessário acrescentar informações no código objeto, tais como a lista de sinais utilizados durante a execução e informações sobre os mesmo.

O código objeto gerado por essa versão do compilador contém a lista de todos os sinais declarados no programa, o autômato e as regras de transição. A lista de todos os sinais declarados no programa, precedida pela palavra reservada DEFINITIONS, conterá as informações da base de dados construída pelo

compilador e aparecerá na parte superior do código objeto. Essa lista tem a seguinte forma:

[m, n, t, f]

onde

- **m** é o módulo a qual pertence o sinal;
- **n** é o nome do sinal;
- **t** é o tipo do sinal e
- **f** é o nome da função de composição associada ao sinal (nil quando não há a função de composição associada).

Um exemplo de como essas definições aparecem no código objeto pode ser observado na Figura 2.

```
DEFINITIONS
[i_, l, Input, nil]
[i_, s, Input, nil]
[i_, gar, Input, nil]
[o_, vazao, Output, nil]
[o_, alarme, Output, nil]
[vazao, teste1, Temporario, nil]
[vazao, teste2, Temporario, nil]
```

Figura 2 - Definição dos sinais no código objeto.

O autômato aparece precedido pela palavra AUTOMATON e possui a seguinte forma:

n s [a]

onde

- **n** é um número ou rótulo que indica o estado;
- **s** é um sinal externo que é esperado pelo estado e
- **[a]** é a regra a ser executada com a chegada desse sinal.

No código objeto, a primeira instrução do autômato sempre tem o estado *init* e não espera por nenhum sinal externo. Por exemplo:

init - [1,*,go_to(1)]

O autômato pode ser melhor entendido através da Figura 3.

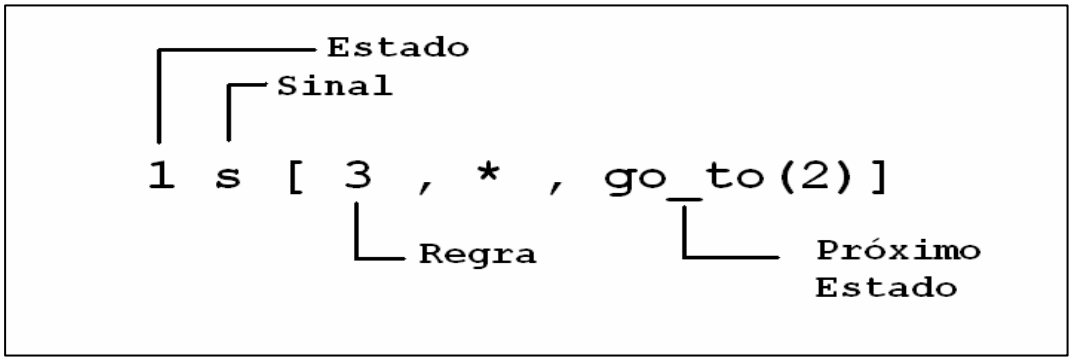

Figura 3 - Funcionamento do autômato gerado.

O exemplo da figura pode ser interpretado da seguinte forma: ao receber o sinal **s**, quando se encontra no estado **1**, o autômato executa a regra **3** e logo após passa para o estado **2**. O asterisco (*) separa regras que devem ser executadas da forma sequencial.

Regras de transição, que complementam o autômato, vêm logo abaixo do autômato. São precedidas pela palavra RULES e são diretamente relacionados com as regras de reação especificadas no código RS. São essas regras que realmente executam as funções do programa. Na sua forma incondicional, são representadas da seguinte forma:

```
n  [sinais] ===> ação
```

onde

- **n** é a identificação da regra a ser executada;
- **[sinais]** é a lista de sinais necessários para disparar a regra, podendo ser uma lista vazia ou conter apenas um sinal externo e/ou vários sinais internos, e
- **ação** é a lista de comandos a serem executados caso a regra seja acionada.

Podem ocorrer também regras condicionais. Estas são representadas por case e aparecem da seguinte forma no código:

n. Case:

n-1. [sinais] {condição$_1$} --->[ação$_1$]

...

n-x. [sinais] {else} --->[ação$_x$]

onde

- **{condição$_n$}** representa uma condição booleana e

- **{else}** indica a ação a ser executada, caso nenhuma condição booleana seja satisfeita. O uso dessa expressão é opcional.

As regras de transição no código objeto podem ser observadas na Figura 4.

```
1.    [] ===> [count:=0]
2.    [] ===> [count:=count+1]
3.    [gar(X)] ===> [count:=count+X,up(teste1(X))]
4.    Case:
4-1.  [teste1(X)] {X>10} ---> [emit(alarme), nl, write(Atenção!)]
```

Figura 4 - Regras de transição no código objeto

3 Ambiente Exec-RS e Interface Gráfica

Neste capítulo, será mostrado, inicialmente, um breve histórico sobre a implementação original de Toscani (1993) para o ambiente de execução de programas RS e de seu compilador, destacando as alterações realizadas por Zschornack (2003) e Oliveira (2005). Em seguida, serão apresentados alguns aspectos relevantes sobre o ambiente de execução Exec-RS e sua interface gráfica, conforme concebidos e implementados por Zschornack e Oliveira.

3.1 Ambiente de execução RS

O ambiente de execução originalmente construído por Toscani (1993), foi implementado na linguagem Prolog e, como conseqüência, a própria linguagem RS foi descrita usando várias expressões inspiradas em sua linguagem hospedeira. Esse ambiente é basicamente composto por dois blocos: um compilador e um simulador da máquina virtual. Para ativação desse ambiente, é necessário possuir um ambiente de programação Prolog, que faça a carga dos módulos que implementam o ambiente de execução RS. Depois disso, para funcionamento do programa, deve ser carregado um arquivo que contenha o código fonte com o programa RS. Na seqüência, são realizadas a compilação e a execução em um fluxo único (uma só operação).

Um fato importante a ser ressaltado é que todos os comandos e interações com o programa são feitos através de linhas de comando, o que torna a interface do ambiente pouco amigável. Para que um usuário possa interagir com o ambiente original da linguagem RS é preciso ter conhecimento dos comandos, que nem sempre são de fácil entendimento. Em alterações seguintes, realizadas neste ambiente, Zschornack (2003) e Oliveira (2005) acrescentaram um novo comando - o

comando ***hlp*** - que imprime uma tabela com todos os comandos utilizados pelo ambiente, fazendo o papel de uma função de ajuda elementar. Outra característica importante era que, ao gerar o código objeto, este era armazenado apenas na memória do compilador, não sendo possível o salvamento em arquivo. Essa característica foi modificada por Zschornack (2003), ao introduzir no ambiente o comando ***sv*** e, assim, tornando possível salvar o código objeto em arquivo, para posterior utilização por outros ambientes. Esta operação só pode ser realizada durante a fase de execução do programa.

O compilador original recebe um arquivo com o código fonte, através do comando ***run*** *E*, onde *E* é o nome do arquivo, e gera um código objeto, passando à fase de execução imediatamente.

Nas modificações feitas por Oliveira (2005) no ambiente já alterado por Zschornack, o código objeto é automaticamente salvo em um arquivo de nome "CodObj.roc" e, posteriormente, durante a execução, é possível salvar este código em arquivo com nome alternativo que também conterá a extensão ".roc", através do comando ***sv***.

No ambiente RS implementado por Toscani (1993), o processo de compilação gera um código objeto, que possui autômato e regras de transição. Após modificações realizadas por Oliveira (2005), esse código objeto passou a conter também a definição dos sinais utilizados pelo programa (ver seção 2.4).

A interface do ambiente RS pode ser observada na Figura 5.

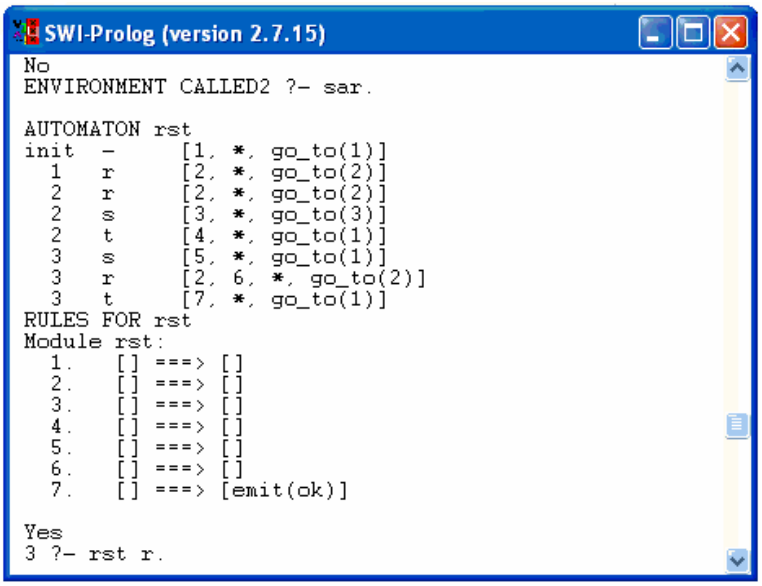

Figura 5 - Ambiente RS.

3.2 Ambiente Exec-RS

Devido ao ambiente de execução RS ser pouco amigável, dificultando a utilização pelo usuário, foi proposta a construção de um novo ambiente de execução para a linguagem, que inclui uma interface gráfica mais amigável, baseada no conceito de WIMP (Windows, Icons, Menus and Pointers). Esse ambiente, denominado Exec-RS, foi proposto e desenvolvido por Zschornack (2003) e, posteriormente, melhorado por Oliveira (2005), utilizando a linguagem Object Pascal, com o ambiente de programação Delphi (CANTÚ, 2000).

O ambiente Exec-RS utiliza o código objeto gerado pelo compilador RS, considerando as alterações apresentadas na seção anterior. É possível gerar o código objeto sem o auxílio do compilador, porém isso é bastante difícil e exige expressiva experiência do programador (ZSCHORNACK, 2003). Esse ambiente pode ser explicado através da Figura 6.

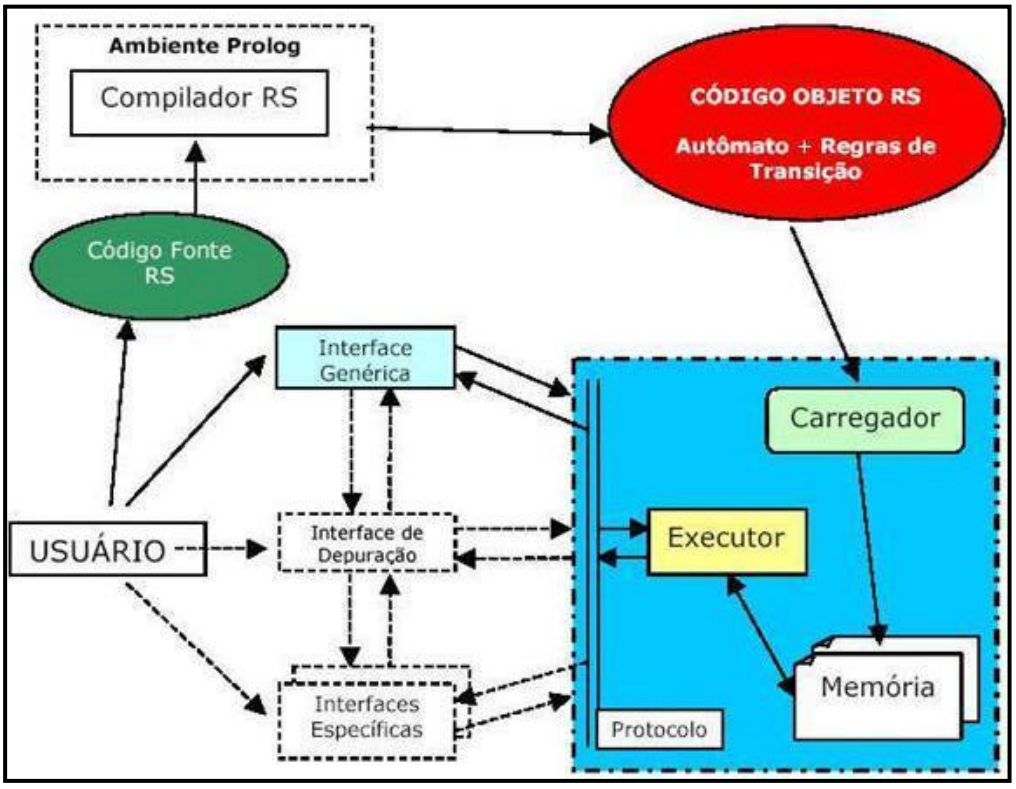

Figura 6 - Esquema de projeto do novo ambiente de execução.
Fonte: ZSCHORNACK, 2003, p.43.

Através da Figura 6, pode ser observado que o ambiente Exec-RS é composto basicamente por três módulos: **carregador, memória e executor**. O

protocolo, que liga o executor à interface, não é necessariamente um módulo. Ele apenas é um conjunto de definições que possibilitam a ligação de qualquer interface ao núcleo, utilizando as informações que este núcleo repasse. Este protocolo de comunicação foi desenvolvido por Zschornack (2003), com o propósito de tornar desnecessárias alterações posteriores, o que foi confirmado por Oliveira (2005), quando realizou diversas alterações no núcleo e na interface genérica, sem precisar realizar alterações no protocolo.

O funcionamento do ambiente pode ser resumido da seguinte forma: o **carregador** recebe um código objeto RS, gerado pelo compilador, e faz um processo de conversão para carregá-lo de forma estruturada na **memória** emulada no ambiente de execução. O **executor** se encarrega de buscar e executar as instruções armazenadas nesta memória, de acordo com os sinais que chegam pela **interface de usuário**, através do **protocolo de comunicação** definido.

O carregador tem três componentes - os analisadores léxico, sintático e semântico. Estes analisadores trabalham de forma a fazer a carga para a memória em um só passo. A memória por sua vez, armazena as informações em forma de tabelas. Na memória, existem quatro tabelas principais, que são: tabela de variáveis, tabela de sinais, tabela de transições do autômato e tabela de regras de transição (OLIVEIRA, 2005).

O ambiente de execução Exec-RS possui quatro modos de execução: modo interativo, modo programado temporizado, modo programado não-temporizado e modo programado depuração.

No modo interativo, o usuário interage com o programa, enviando os sinais de entrada ao núcleo reativo, de forma aleatória, sem preocupação de tempo, a partir de elementos de interação projetados na interface, tais como os botões que se encontram mais à esquerda, na interface genérica (Figura 9). No modo programado temporizado, o usuário programa os sinais que deseja enviar nos intervalos de tempo desejados. Diferentemente do modo anterior, o modo programado não-temporizado ignora o fator tempo e envia sinais um após o outro. No modo depuração, o usuário controla os intervalos de tempo entre os sinais programados, para que possa verificar o comportamento do programa, através de valores de sinais e variáveis, por exemplo.

A escolha do modo de execução é realizada através da janela que é mostrada na Figura 7 a seguir.

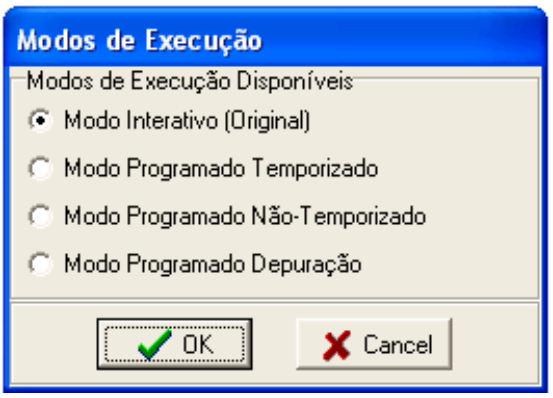

Figura 7 - Modos de Execução.

3.3 Interface gráfica

Um dos principais motivos da criação do ambiente Exec-RS foi a melhoria da forma de interação entre o usuário e o ambiente de execução da linguagem RS, pois o ambiente original é de difícil utilização, tendo em vista que o usuário precisa conhecer todas as linhas de comando. Diferentemente da interface do Exec-RS, que possui ícones e botões que tornam os comandos dedutíveis para o usuário, essa interface possui formato genérico, sendo que qualquer programa que seja carregado e executado no ambiente terá sempre o mesmo formato. A interface do Exec-RS pode ser dividida em dois grupos: interface de carga e interface de execução (ZSCHORNACK, 2003).

A interface de carga é a interface que é vista ao abrir o programa Exec-RS, e é responsável por:

- carga do arquivo contendo o código objeto;
- configuração da camada de manipulação de dados e
- visualização do conteúdo da memória e erros ocorridos durante a carga.

Esta interface pode ser observada na **Erro! Fonte de referência não encontrada.** a seguir.

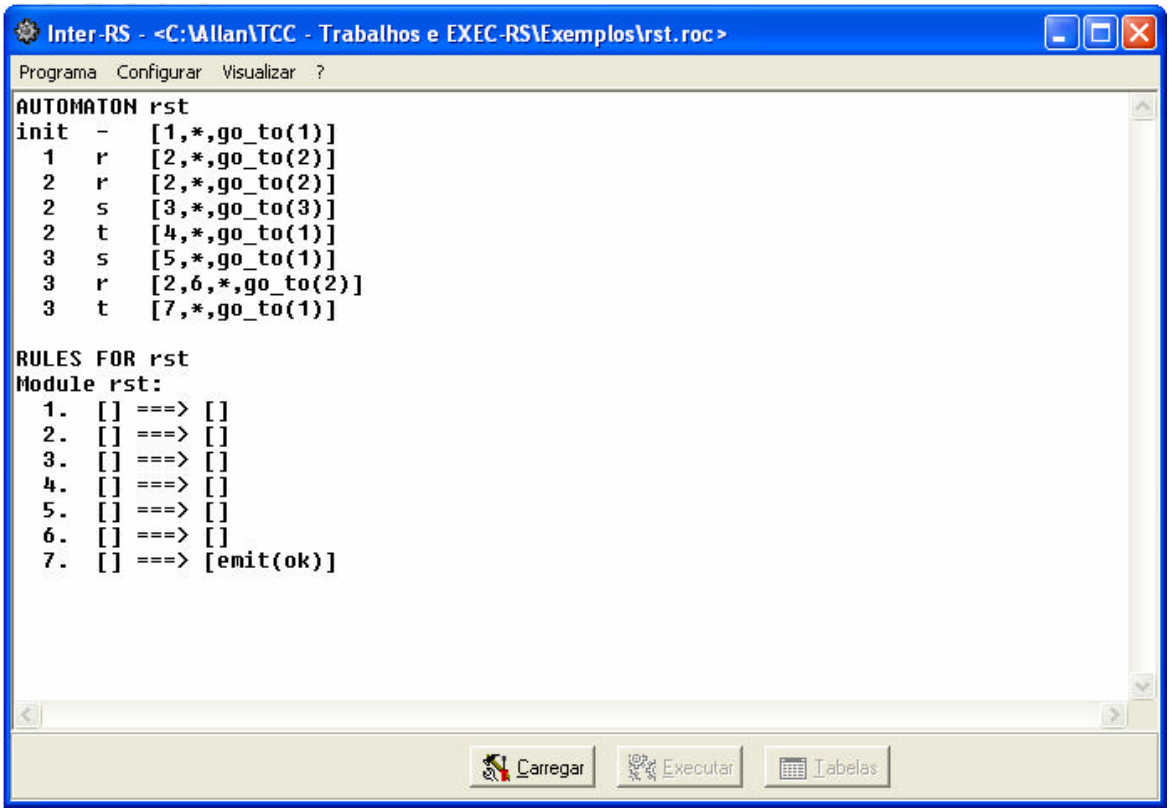

Figura 8 - Interface de Carga

A interface de execução é responsável pela interação do usuário com o programa, pois nela são realizadas as entradas de sinais e dados necessários ao programa, a visualização de sinais de saída e também a escolha dos modos de operação.

A interface gráfica proposta por Zschornack (2003), utiliza um protocolo de comunicação entre o núcleo reativo e o exterior. Atualmente, o ambiente Exec-RS possui somente uma interface genérica, representada por uma janela que se abre ao executar o programa RS, e com isso não é possível perceber todo o potencial do protocolo de comunicação definido, pois este protocolo tem a finalidade de permitir à criação de interfaces diversas (interfaces específicas para programas típicos da linguagem), tornando mais agradável a utilização do ambiente.

Na janela da interface genérica, encontram-se várias informações do programa que está sendo executado, como as tabelas de sinais e variáveis, bem como mensagens e sinais emitidos, além do estado atual em que se encontra o programa (Figura 9).

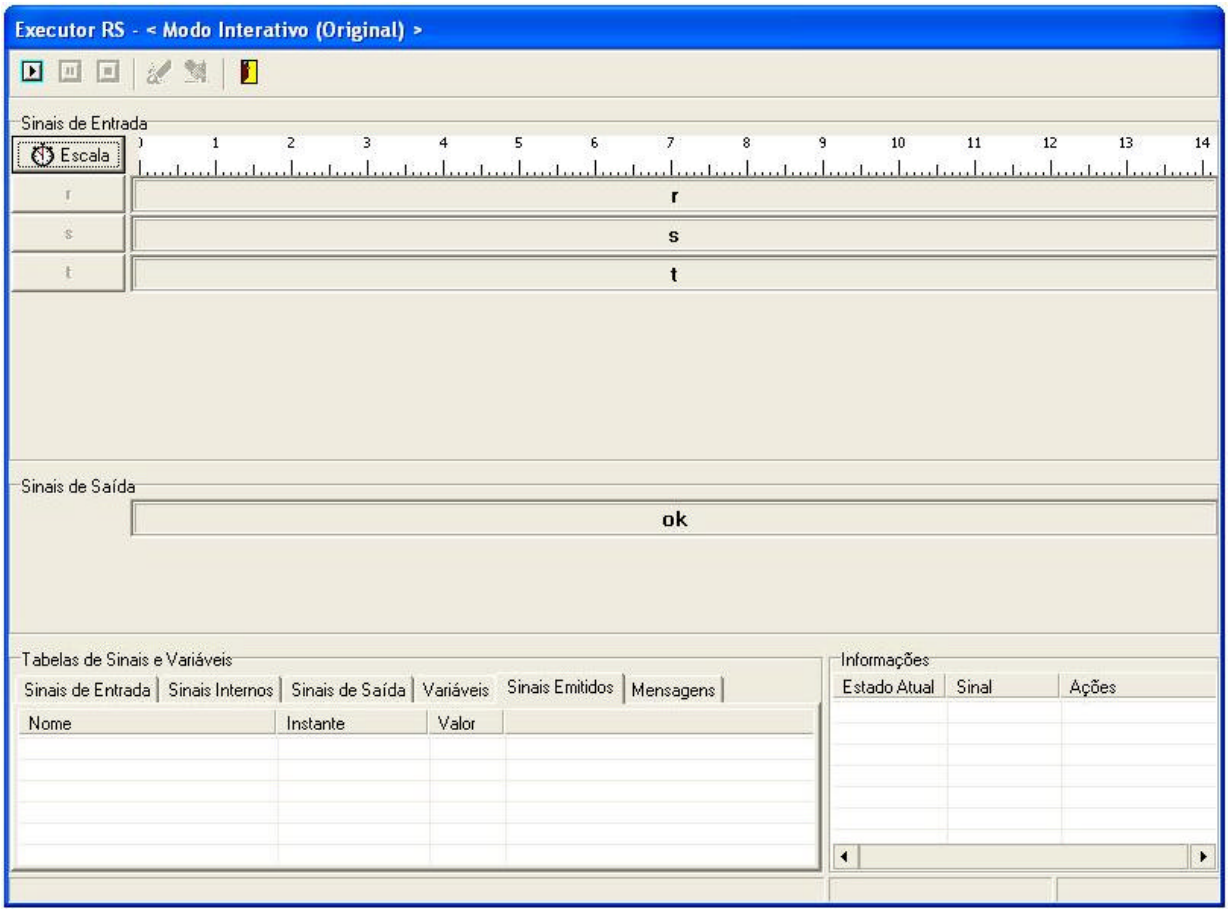

Figura 9 - Interface genérica.

4 Tratamento de Exceções na Linguagem RS

A linguagem RS define o tratamento de exceções a partir de sua versão plena. O poder desta versão da linguagem decorre simplesmente da complementação da versão RS-2 com o acréscimo dos mecanismos que implementam o tratamento de exceção. A seguir, serão apresentados estes mecanismos, através de exemplos e descrições de suas características no código fonte e no código objeto, destacando as implicações nos autômatos e nas regras de transição.

4.1 Tratamento de exceções

Exceções são condições especiais no sistema reativo que causam mudanças bruscas no seu estado (TOSCANI, 1993). Essas condições especiais são sinais que, sinalizados e detectados em determinados momentos da execução do sistema reativo, receberão tratamento especial. Estes sinais podem ser de dois tipos:

- sinais que ocorrem diretamente do exterior e são detectados pelo núcleo reativo;

- sinais que são produzidos e tratados internamente.

As mudanças que ocorrem no programa, ao serem detectadas exceções podem ser descritas da seguinte forma:

- Ocorrerá um “reset” nos módulos envolvidos, ou seja, sinais internos serão desligados e caixas de regras desativadas.

- Novos estados serão definidos pela regra de exceção.

As condições de exceções no código fonte são descritas pelos chamados eventos de exceção. Esses eventos podem representar sinais puros ou sinais que

contenham informações arbitrárias. No código fonte, os eventos de exceção são descritos através do bloco on_exception que tem a seguinte forma:

```
On_exception:
[evt1✍act1;
 ...
 evtp✍actp;          (p>=0)
]
```

Cada evento “evt” tem sua ação “act” correspondente. As ações das regras podem assumir duas formas, qualquer uma delas sendo concluída por uma cláusula reset (que é opcional, ou seja, só é aplicável se a lista de pares m#e, que significa módulo e evento, não é vazia):

```
[c1,c2,c3,...,cn] reset [m1#e1,...,ml#el]        (n>=0, l>=0)
```

ou

```
case
[evt1✍[c11,c12,...,c1n1],
 ...,
 evtk✍[ck1,ck2,...,cknk]
] reset [m1#e1, ..., ml#el]      (k>=2, l>=0)
```

onde cada opção especifica um evento e uma lista de comandos, a qual pode ser vazia.

4.1.1 Reset

O sufixo opcional *reset* (não é especificado quando a lista de pares m#e é vazia) permite que sejam propagadas mudanças através dos módulos. Para exemplificar a regra reset, pegamos o seguinte exemplo:

```
S✍[emit(A)] reset [mod2#evt]
```

Supondo-se que esta regra esteja definida no módulo mod1, e a regra de exceção seja ativada pelo sinal S, o módulo mod2 sofrerá reset simultaneamente

com o módulo mod1, e logo após será executado o evento evt do módulo mod2 paralelamente com a emissão do sinal A.

Para demonstrar como um módulo *m* sofre alteração através da opção *b(X,Y)* do evento *a*, o sufixo deve aparecer da seguinte maneira, como ilustra a Figura 10.

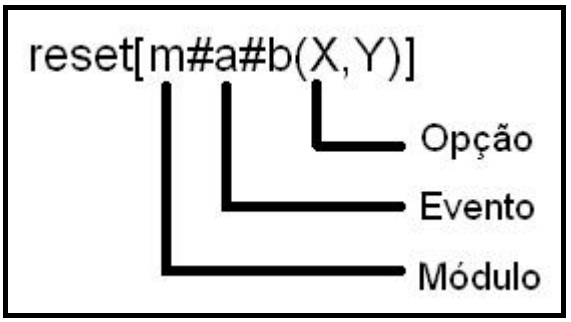

Figura 10 - Reset

4.1.2 Regras *case* em eventos de exceção

Quando se deseja representar múltiplas condições de uma regra de exceção, então se utiliza a regra *case.* Cada uma das condições da exceção é representada por uma opção da regra. Nestes casos, a condição de disparo da regra deve ser um evento puro (que não possui valor), e os eventos das opções ou sinais de guarda, podem ser ou não, sinais puros (TOSCANI, 1993). A condição de disparo é representada pelo sinal mais à esquerda na regra, enquanto os sinais de guarda são os sinais que representam as opções. A Figura 11 abaixo demonstra como são dispostos estes sinais nas regras *case*, conforme devem ser utilizados no código fonte, onde *trap* é o evento de exceção e *c1* e *c2* são opções do evento.

```
On_exception:
[ trap ===> case [
            c1 ---> [nl, write(trap#c1)],
            c2 ---> [nl, write(trap#c2)]
             ] reset[ ]
],
```

Figura 11 - Opção case nas regras de exceção.

Um outro exemplo demonstrado pela Figura 12 mostra como a regra *case* fica nas regras de reação do código objeto.

```
2.   Excp case "trap" (reset []):
2-1. [c1] ---> [nl, write(trap # c1)]
2-2. [c2] ---> [nl, write(trap # c2)]
```

Figura 12 - Regra case no código objeto.

Neste exemplo, a regra é disparada pelo sinal “trap” e as opções são selecionadas pelos sinais de guarda “c1” e “c2”. Essas chamadas da regra *case* são feitas da seguinte maneira dentro do código:

```
raise(trap#c1)],
```

Esta é a maneira para efetuar chamadas do mecanismo de exceção, tanto no código fonte como no código objeto.

As regras do tipo *case* representam múltiplas condições de exceção relacionadas entre si e mutuamente exclusivas, ou seja, no caso de mais de uma opção ser verdadeira, apenas a primeira opção encontrada será executada, sendo que cada condição é representada por uma opção da regra. Por isso as opções que ativam as regras devem ser diferentes entre si (TOSCANI, 1993).

4.1.3 Sinalização de eventos

Durante a execução do programa, uma condição de exceção pode ser sinalizada pelo comando *raise*. Esse comando especifica um evento dentre os que estão declarados no bloco *on_exception* (TOSCANI, 1993). O evento especificado por esse comando pode aparecer de diversas formas. Por exemplo:

Raise(*error*) - especifica o evento *error* para o bloco *on_exception*;

Raise(b(X)) - especifica o evento b com o parâmetro X para o tratamento dessa exceção;

Raise(e#op) - o comando representa a opção op do evento e.

4.1.4 Sinalização de eventos externos e internos

Como citado anteriormente, os eventos de exceção podem ocorrer através de sinais emitidos e detectados internamente ou através de sinais vindos diretamente do exterior. Os eventos de exceção internos são ativados com o comando *raise*, enquanto na sinalização de eventos externa isso é desnecessário. Para simplificar a programação, no caso de eventos vindos do exterior, basta definir o sinal externo que se deseja tratar como evento de exceção, diretamente no bloco *on_exception*. Sendo assim, sinais desse tipo estarão sempre habilitados e aparecerão em todos os estados do autômato (TOSCANI, 1993). No autômato, existirá o estado especial *excp*, que complementará todos os outros estados, ou seja, *excp* estará sempre habilitado à espera de sinal (OLIVEIRA, 2005) e também poderá ser o único estado à espera de sinal.

4.1.5 Sinal *Other*

Uma facilidade que o mecanismo de tratamento de exceção possibilita, é o tratamento uniforme de sinais. Essa facilidade permite que qualquer sinal externo que ocorra de forma inesperada, seja tratado. No código fonte, esse tipo de tratamento será explicitado pelo nome reservado *other* (como são nomes reservados, não poderão aparecer nas declarações de *input*). Onde se encontrar a representação *other*, qualquer sinal externo que for detectado, receberá o mesmo tratamento. Para ilustrar o funcionamento dessa facilidade, pode-se imaginar um

programa que possui os sinais de entrada A, B, C, D e E e possui uma caixa de regras com a seguinte estrutura:

```
Box_n:
[ A ==> [emit(ok)],
  B ==> [emit(ok)],
  Other ==> [emit(erro)]
]
```

No caso dessa caixa receber os sinais externos C, D ou E, o programa emitirá o sinal *erro.*

No código objeto, o sinal *other* será representado da seguinte forma, no autômato:

```
2 other [2,*, go_to(1)]
```

4.1.6 Representação dos mecanismos de exceção no código objeto

Os autômatos e regras de transição correspondentes aos tratamentos de exceção têm formas especiais de serem representados no código objeto. A seguir pode ser visto na Figura 13, alguns exemplos dessas representações.

```
5. Excp rule (reset [ ]):
   [ ] ==> [emit(red_on), emit(go_off)]

2. Excp rule (reset [m2#trap#t1(A)]):
   [excp(A)] ===> [nl, write(excp(A))]

2.   Excp Case "trap" (reset [ ]):
2-1. [c1] ---> [nl, write(trap#c1)]
2-2. [c2] ---> [nl, write(trap#c2)]

8. Excp Case "Trap" (reset [ ]):
8-1. [t1(D)] ---> [nl, write(trap(t1(D)))]
8-2. [t2] ---> [nl, write(trap(t2))]
```

Figura 13 - Exemplos de mecanismos de exceção no código objeto.
Fonte: OLIVEIRA, 2005, p. 38

A regra poderá também aparecer da seguinte maneira:

```
3. Excp rule (reset[ ]):
          [ ] ==>[ ]
```

Neste caso, não haverá regras a serem executadas e o módulo sofrerá *reset* e não serão executadas outras ações.

As facilidades de sinalização de exceções através de sinais externos e do sinal *other* são refletidos no autômato (OLIVEIRA, 2005) e são observadas nos exemplos:

```
init     -    [1,*,go_to(excp)]
excp  coin    [2,*,go_to(1)]
5     other   [5,*,go_to(1)]
```

5 A Inclusão dos Mecanismos de Tratamento de Exceções (RS-Plena) no Ambiente Exec-RS

Neste capítulo, são apresentados aspectos sobre as implementações realizadas para a integração dos mecanismos de tratamento de exceções da linguagem RS (RS-Plena) no ambiente Exec-RS. Também são tratadas as alterações necessárias para a consolidação do ambiente, visando a inclusão dos novos recursos e as correções de erros detectados na versão anterior do ambiente.

5.1 Análise de requisitos

O estudo detalhado do código objeto da linguagem e do ambiente Exec-RS, na forma final apresentada por Oliveira (2005), permitiu estabelecer uma relação clara entre as características dos mecanismos de tratamento de exceção e as estratégias adotadas naquela versão, o que resultou em uma lista de tarefas relacionadas às necessidades de alterações e ampliações para a consolidação do ambiente.

Constatou-se que as estruturas de dados definidas no programa já se encontravam num estágio adequado à inclusão dos novos recursos, podendo ser consideradas prontas, sem exigir modificações.

Foram necessários alguns ajustes no ambiente de execução, em virtude de erros detectados durante a ampliação do ambiente e atributos que poderiam ser melhorados, mas nada que alterasse a funcionalidade do ambiente.

Para a consolidação da linguagem RS, foram estudados os módulos de implementação do programa que define o ambiente de execução, verificando-se que as implementações e alterações necessárias para o tratamento de exceções deveriam ocorrer nas unidades "*executor*" e "*transicoes*".

À medida que as implementações da ampliação do ambiente iam ocorrendo, foi observada a necessidade de ajustes em diversos outros módulos do programa.

Com essas implementações o ambiente tornou-se capaz de aceitar autômatos e regras de códigos objetos, como o representado na Figura 14, que possui todos os mecanismos de exceção. Ao longo do capítulo serão utilizados trechos desse código, como exemplo.

```
AUTOMATON e
init  -     [1, 8, *, go_to(3)]
excp  a     [2, *, go_to(1)]
  1   u     [5, *, 4, 9-2, *, go_to(2)]
  1   s     [6, *, 3-1, 9-1, *, go_to(3)]
  1   t     [7, *, 3-2, 9-1, *, go_to(3)]
  2   s     [10, *, go_to(2)]
  2   other [11, *, go_to(2)]
  3   u     [5, *, 4, 9-2, *, go_to(2)]
  3   t     [7, *, 3-2, 9-1, *, go_to(3)]
  3   s     [6, 10, *, 3-1, 9-1, *, go_to(3)]
  3   other [11, *, go_to(3)]
RULES FOR e
Module m1:
  1.   [] ===> []
  2.   Excp rule (reset []):
       [] ===> [nl, write(Evento Sinalizado do Exterior!!)]
  3.   Excp case "trap" (reset [m2#w#a1]):
  3-1. [c1] ---> [nl, write(trap#c1)]
  3-2. [c2] ---> [nl, write(trap#c2)]
  4.   Excp rule (reset [m2#w#a2]):
       [] ===> [write(trap2)]
  5.   [] ===> [nl, write(uuu)]
  6.   [] ===> [nl, write(m1 s), raise(trap#c1)]
  7.   [] ===> [nl, write(ttt), raise(trap#c2)]
Module m2:
  8.   [] ===> []
  9.   Excp case "w" (reset []):
  9-1. [a1] ---> [write(rrr)]
  9-2. [a2] ---> [write(rrr t2)]
 10.   [] ===> [nl, write(m2 s)]
```

Figura 14 - Exemplo de código objeto com linguagem Plena.

5.2 Implementação

Após a avaliação das funcionalidades e do funcionamento da versão anterior do programa e a análise dos requisitos, foram estabelecidas as estratégias de desenvolvimento das funções correspondentes aos novos recursos (da última versão da linguagem RS), cujas etapas de implementação serão detalhadas nas seções seguintes.

5.2.1 O comando Raise

O comando *raise* é implementado conforme sua descrição realizada na seção 4.1.3. A unidade *executor* já estava modelada para realizar suas funções através do procedimento *ExecRaising*, o qual extrai, da própria instrução armazenada no código objeto, o parâmetro contendo os sinais indicados.

O procedimento *ExecRaising* é chamado diretamente pelo procedimento *ExecutaInstr,* o qual encaminha a execução de cada instrução encontrada no código objeto.

O comando *raise* pode ser comparado ao comando *up* da linguagem RS (TOSCANI, 1993). Sendo assim, a sua implementação dentro do executor é bastante similar a do procedimento *ExecUp*, que também se encontra dentro do módulo *executor*. Estas funções ativam os sinais internos que recebem como parâmetros. A função *ExecRaising,* se necessário, separa as partes da instrução e habilita o sinal interno a ser utilizado.

Considere-se a execução do comando **`raise(trap#c1)]`**, conforme usado na regra 6 do exemplo sugerido (Figura 14). A função *ExecRaising* recebe como parâmetros **`trap#c1`** e separa os dois sinais (trap e c1), fazendo a ativação dos mesmos para que possam ser interpretados na continuação da execução do programa.

5.2.2 O comando Reset

A modelagem do comando *Reset* estava preparada, no ambiente Exec-RS, de forma similar à utilizada para o comando *Raise*. Entretanto, foi necessária a adoção de uma estratégia diferente, considerando que as estruturas da tabela de

regras já possuem campos onde são armazenadas as informações necessárias para a execução deste comando (OLIVEIRA, 2005, p.47).

Novamente, tomando o exemplo da Figura 14, considere-se a regra de número 3, que contém uma instrução *reset* (**`reset[m2#w#a1]`**). Os parâmetros passados pelo comando *reset*, representados por **`m2#w#a1`**, são armazenados no campo da tabela de regras destinado para isso.

Observa-se que a função *reset* pode aparecer, dentro do código objeto, em regras compostas, ou seja, uma regra *i* que possui regras componentes, como *i-1*, *i-2* etc. Sendo assim, na sua execução dentro dos autômatos do código objeto, a regra principal *i* nunca é executada, apenas uma de suas derivações. Isto caracteriza um problema para o tratamento, pois o comando *reset* aparece vinculado à regra principal. Assim, tornou-se necessário determinar que todas as regras compostas sejam sempre verificadas para analisar a existência de parâmetros dentro da estrutura *reset* das tabelas. No caso destes campos de reset possuírem informações, as medidas necessárias deverão ser tomadas.

Para ilustrar como isso ocorre, o exemplo a seguir (extraído da Figura 14) mostra como essas regras compostas aparecem no código objeto:

```
3.   Excp case "trap" (reset [m2#w#a1]):
3-1. [c1] ---> [nl, write(trap#c1)]
3-2. [c2] ---> [nl, write(trap#c2)]
```

Com isso, o *reset* passou a ser tratado dentro da função *ExecutaRegra* do módulo *executor*, sendo que a chamada desta função no ambiente ocorre antes da função de execução *ExecutaInstr*. O funcionamento do comando *reset* também é semelhante ao funcionamento dos comandos *up* e *raise*, pois deve ativar sinais internos do código objeto, os quais determinarão ações em outras partes do programa.

5.2.3 Eventos sinalizados do exterior

Os eventos sinalizados do exterior exigiram alterações na função *Obter* da unidade *transicoes*, que determina as ações correspondentes a uma combinação de estado e sinal. Esta função é usada para relacionar o estado atual em que o autômato RS se encontra, com o sinal de entrada emitido. Porém, ela não estava

preparada para verificação do estado especial *excp*, que, no caso de ser encontrado em um autômato RS, está sempre habilitado a receber sinais. Para isso é necessário que, a cada chamada desta função, seja verificada a existência deste estado especial no código objeto e relacioná-lo com o sinal de entrada emitido. Este estado especial encontra-se no autômato do código objeto, da seguinte forma:

```
excp  a  [2, *, go_to(1)]
```

5.2.4 Tratamento de sinais inesperados (*other*)

O tratamento de sinais inesperados na linguagem RS – Plena, assim como o tratamento de eventos sinalizados do exterior, deve ocorrer dentro da função *Obter* da unidade *transicoes*. Assim como esta função relaciona o sinal externo emitido com o estado atual do programa RS, ela deve preocupar-se no caso de não haver relação entre estado e sinal emitido. Caso esta relação não exista, a função deve realizar a verificação da presença do sinal *other* no programa RS relacionado com o estado atual em que se encontra o programa. Caso exista, as devidas medidas serão executadas.

O exemplo (Figura 14) mostra como o sinal *other* aparece no autômato RS:

```
excp  a     [2, *, go_to(1)]
2   s     [10, *, go_to(2)]
2   other [11, *, go_to(2)]
```

Caso o autômato encontre-se no estado **2** e receba um sinal diferente de **s** ou **a** os testes descritos anteriormente serão executados, caracterizando o tratamento uniforme dos sinais.

5.2.5 Outras modificações

Algumas modificações no executor foram necessárias para melhoria do ambiente ou correção de pequenos erros. Os seguintes quesitos foram alvos destas alterações:

a) na escolha das extensões de arquivos a serem aberto pelo executor, as opções de tipos de arquivos referenciavam a extensão ".ric", quando, na verdade, esta extensão deveria ser ".roc" (*reactive object code*);

b) Limpeza das tabelas e das mensagens emitidas e eliminação dos botões - estes elementos que aparecem na interface genérica não eram limpos (reinicializados) ao término da execução de um programa, mantendo informações indesejadas para a próxima execução de programas RS; para a limpeza das tabelas e eliminação dos botões criados anteriormente, a solução encontrada foi realizá-las antes da carga de um arquivo, na função *BtnCarregarClick* da unidade *principal*; no caso das mensagens emitidas, a solução utilizada foi a execução do método *Clear* do objeto correspondente na unidade ViewExec, quando se fecha a interface;

c) no caso de códigos objetos incorretos, a impressão de erros na tela era acumulada a cada tentativa de execução do programa, pois a tabela de erros não era limpa, o que foi resolvido com o uso do método *Clear* do objeto no ato do fechamento da caixa que imprimia os erros na tela;

d) em programas RS que continham tratamento inesperado de sinais, a interface genérica criava um botão para a emissão do sinal "*other*", o que é incorreto, pois esse sinal não deve ser emitido pelo usuário; esse detalhe foi corrigido na unidade *ViewExec*, alterando o procedimento *PanelSinais*, que é o responsável pela criação dos botões;

e) o tratamento das expressões em comandos de atribuição apresentava uma falha, fazendo com que o valor de uma variável fosse substituído por seu nome, em determinadas circunstâncias, ocasionando erros na execução, o que foi corrigido dentro da função *Opera* da unidade *Executor*;

f) a função *RegraHabilitada* no módulo *executor* foi alterada, para garantir o funcionamento correto da ocorrência da emissão de um mesmo sinal mais de uma vez na mesma reação, prevista na linguagem RS-1, pois as pseudo-variáveis (campos de valores dos sinais) usadas na habilitação das regras não estavam sendo atualizadas em alguns casos; essas pseudo-variáveis somente eram atualizadas caso a condição de disparo da regra fosse um sinal de entrada, ignorando a existência de casos em que as condições de disparo são sinais emitidos pelo programa.

6 Desenvolvimento de Interfaces Demonstrativas

Este capítulo apresenta as interfaces específicas implementadas para alguns programas-exemplos, com o objetivo de validação e demonstração do protocolo de comunicação. As interfaces foram construídas em novos módulos que foram incorporadas ao ambiente de execução.

Para a implementação destas interfaces particulares, além da implementação de novas unidades, que modelam essas interfaces, foram necessárias, também, mudanças na unidade *AmbExec*. Nesta unidade foram acrescentados testes para verificar a existência de tratamento específico para o programa objeto submetido à carga. Na unidade *principal*, ao carregar o arquivo que contém o código objeto, é capturado o nome deste arquivo para ser utilizado nesta verificação. Caso exista alguma interface definida com o mesmo nome, ela será assumida na execução do programa, caso contrário, será utilizada a interface genérica.

O desenvolvimento das interfaces foi realizado, respeitando integralmente o protocolo de comunicação entre a interface e o executor, conforme definido por Zschornack (2003). As interfaces foram desenvolvidas em grau crescente de dificuldade, ou seja, primeiramente, as interfaces de programas mais simples e, posteriormente, a criação de interfaces para programas mais complexos. Na construção destas interfaces não foram observados quaisquer preceitos metodológicos relacionados às técnicas de interação humano-computador, pois o objetivo de sua definição é especificamente a técnica de comunicação dos módulos de interface com o módulo de controle.

O protocolo de comunicação não é caracterizado por um módulo, mas sim por funções que fazem a comunicação da unidade de execução com a unidade de interface. Dentro das unidades de interface são declarados objetos que tornam

possível fazer chamadas de métodos que caracterizam o protocolo de comunicação. A interface é inicializada no módulo *AmbExec* através do procedimento *InitInterface* que recebe como parâmetros os objetos tipo *memoria* e *executor*.

Nas seções seguintes, são feitas as apresentações das interfaces específicas que foram implementadas.

6.1 Interface RST

A primeira interface a ser totalmente concluída foi a do programa RST, que é bastante simples do ponto de vista da linguagem (Figura 15). O programa emite o sinal "**ok**", quando estimulado corretamente pela seqüência "**r**", "**s**" e "**t**". Esta interface possui somente o tipo de execução interativa. O módulo que a define recebe o nome *RST* (o mesmo nome do programa).

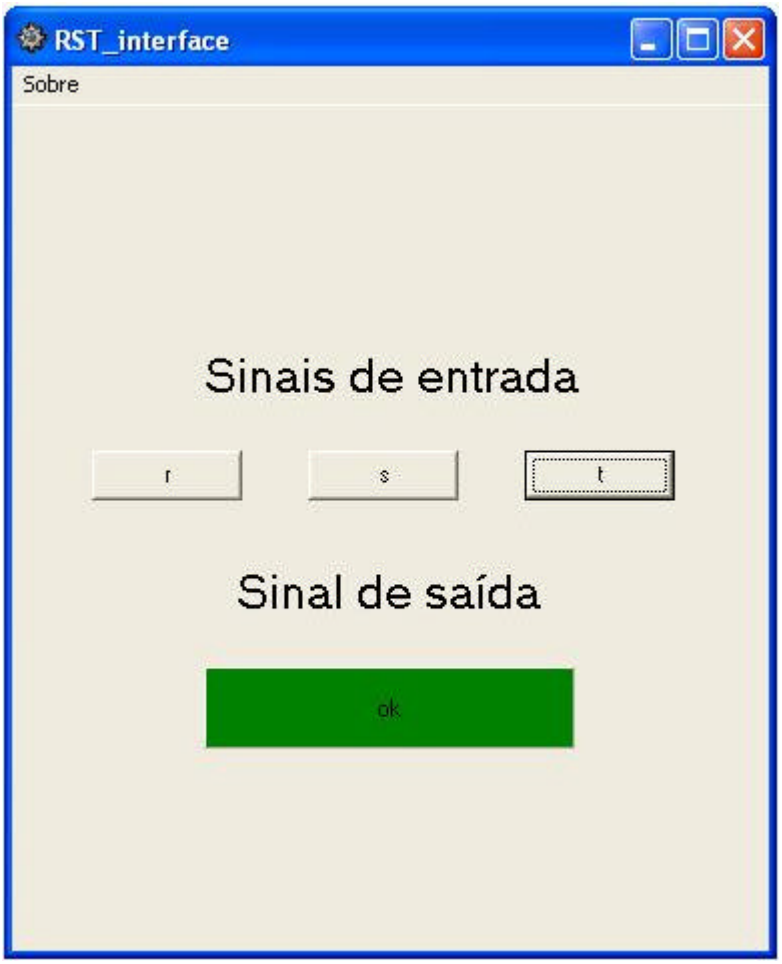

Figura 15 - Interface específica do programa RST.

Os testes de como deveria ser feita a utilização do protocolo de comunicação foram realizadas nesta unidade, que se baseou na unidade *ViewExec*, onde se encontram os procedimentos para a interface genérica.

6.2 Interface do Jogo do Reflexo

Após a conclusão da interface RST, partiu-se para a confecção de interfaces de programas com maior complexidade. O exemplo do jogo do reflexo foi utilizado logo a seguir, pois esse programa, além de ser um exemplo mais realista, possui um maior número de autômatos e regras além de possuir dois modos de execução simultâneos, caracterizando os modos de execução interativo e programado temporizado, pois esta aplicação deve receber estímulos externos em tempos pré-estabelecidos e estímulos externos do usuário. Este jogo mede o tempo de reflexo que o usuário leva para pressionar a tecla correta, quando o programa emitir o sinal de permissão.

O jogo começa com o usuário inserindo um valor (***coin***) para estabelecer quantas vezes irá jogar. Logo após, o programa começa a ser estimulado com sinais externos que são emitidos em tempos estabelecidos. O jogador deve pressionar a tecla "**ready**" para indicar que está pronto para começar a jogar. O painel do jogo indica ao jogador através do indicador "**go**", que ficará verde neste momento, que já poderá apertar a tecla "**stop**" para medir o seu tempo de reflexo. Se o jogador não entrar com sinais em um determinado tempo, será considerado abandono do jogo e, no caso de teclas serem pressionadas em momentos errados, o programa emitirá os devidos sinais, que na interface são representados por sons.

Esta interface possui características dos modos de execução interativo e programado temporizado, pois o usuário deve interagir com a aplicação, e a aplicação deve receber contínuos estímulos, em determinados estados que se encontra o programa, para contagem do tempo. Esta aplicação também possui o uso de mecanismos de exceções além de um procedimento externo, que foi implementado na linguagem de programação C. Este procedimento gera um número aleatório que será utilizado para estipular o tempo de espera para emissão do sinal "**go**". Pequenas alterações no código objeto do jogo do reflexo tiveram que ser feitas, para ajustar o "tick" do código com o tempo medido.

A Figura 16 mostra a interface do jogo do reflexo. Esta interface foi gerada com o nome de *reflexo*, pelas razões já explicadas

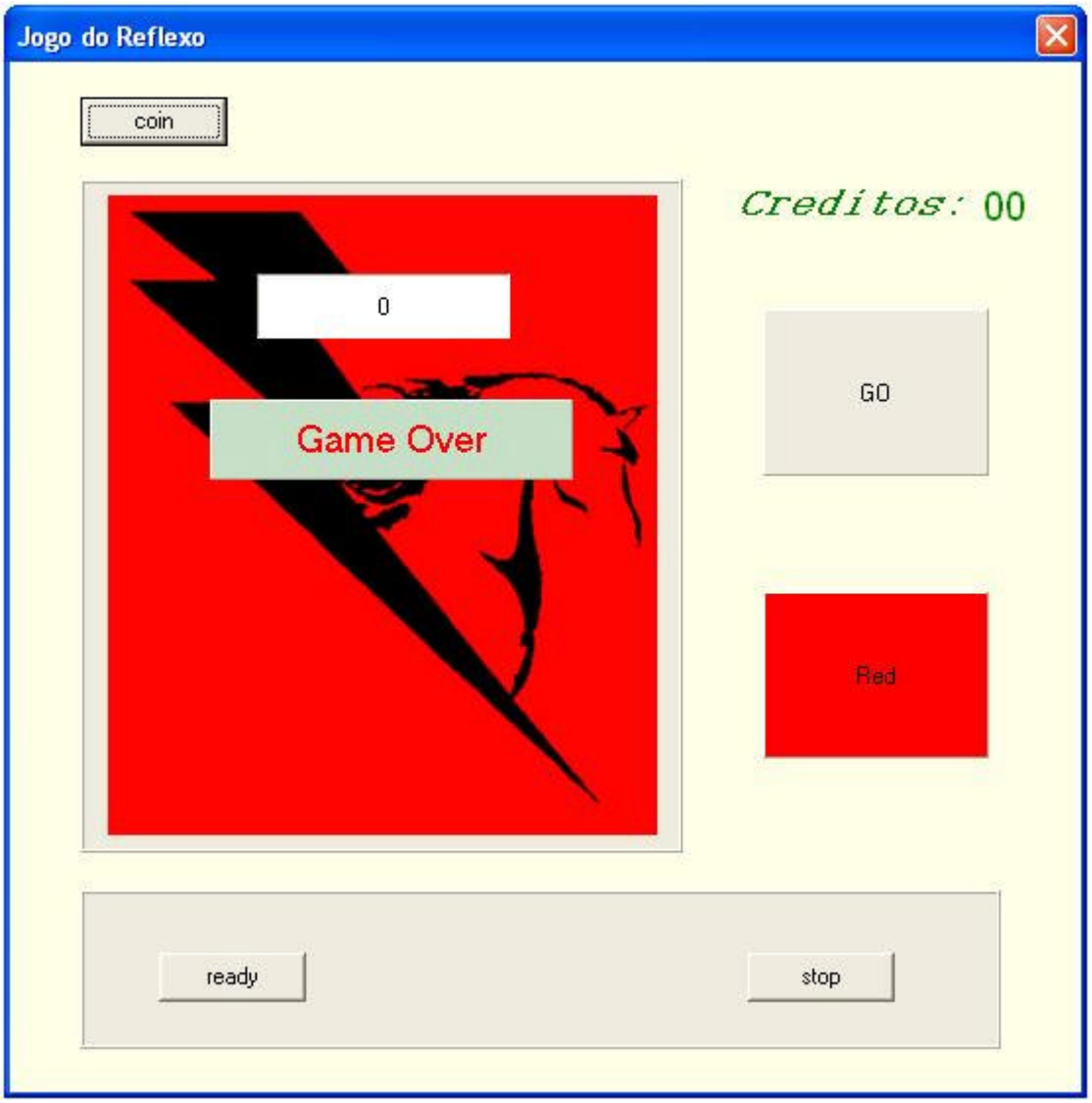

Figura 16 - Jogo do reflexo

6.3 Interface do Relógio Digital

Outra interface específica produzida para o ambiente Exec-RS foi a interface do relógio digital. Esta aplicação foi anteriormente utilizada para demonstrar os formalismos das linguagens Statecharts e Esterel e, posteriormente, utilizada para ilustrações da linguagem RS (TOSCANI, 1993).

O controlador do relógio é uma aplicação de utilidade real e que possui problemas não triviais. Essa aplicação possui cinco módulos e inúmeros procedimentos externos. A interface gráfica do relógio possui quatro botões, cujas funções são distintas, dependendo do modo em que se encontra a aplicação. Existem quatro botões previstos por Toscani (1993), que são: **ul** (*upper left button*), **ll** (*lower left button*), **ur** (*upper right button*) e **lr** (*lower right button*). Além destes botões, a interface possui mais um botão que tem o nome de **s** (segundos), que foi acrescentado para testes e melhor compreensão da aplicação. Com isso, a aplicação ganhou dois modos de execução: modo automático, onde o sinal de segundo entra em tempos pré-determinados e o botão **s** permanece desabilitado, e modo manual, onde o usuário deve entrar com o sinal **s** através do botão. Esses modos foram acrescentados e não estavam previstos na interface definida por Toscani.

Pode-se observar, portanto, que no modo automático a interface possui dois modos de execução, modo programado, pois recebe sinais em intervalos de tempo constante, e modo interativo, pois o usuário pode emitir sinais para a aplicação. No modo manual, a aplicação é puramente interativa.

A interface possui *displays* que sinalizam as horas e indicam a data corrente, além de indicadores de alarme. A interface e os procedimentos externos implementados possibilitam o funcionamento correto dos dois modos básicos de uso do relógio: modo *watch* e modo *set_watch*.

Nesta implementação, a interface do relógio digital apresenta três dos cinco módulos previstos: o módulo *display* (controlador de mostrador), o módulo *watch* (normal) e o módulo *button* (interpretador de estímulos). Estes módulos apresentam suas próprias características, como sinais internos e variáveis, além de procedimentos externos utilizados.

Para o funcionamento correto das funções que este exemplo está apto a executar, foram necessárias implementações de procedimentos externos. Para isso foi utilizada a linguagem de programação C. Estes procedimentos funcionam com a passagem de parâmetros conforme definida por Oliveira (2005).

Para a confecção desses procedimentos, depois de conhecido como ocorre a passagem de parâmetros nos procedimentos externos, foi necessário também definir como seriam tratadas as informações referentes ao tempo nestes procedimentos. Foi estipulado então a forma HHMMSSdDDmm, que contém as

informações de tempo necessárias para a contagem do tempo, e que possuem como significado o seguinte:

- HH - representa a hora;
- MM - representa os minutos;
- SS - representa os segundos;
- d - representa o dia da semana;
- DD - refere-se ao dia do mês;
- mm - representa o mês corrente.

Com isso, os procedimentos externos dão o tratamento adequado de modo a marcar o tempo e devolver valores que devem ser interpretados e tratados adequadamente pela interface.

Os procedimentos utilizados são:

- compare_A_T_to_W_T;
- get_watch_beep_val;
- incr_W_T;
- incr_W_T_in_set_M;
- set_W_T;
- toggle_24_M_in_W_T;
- toggle_24H_M_in_A_T;
- watch_Date_to_mini_D;
- Watch_Day_to_alph_D;
- watch_T_to_main_D.

Existem também cinco modos de operação para o relógio, sendo que dois deles serão apresentados nesta seção, o modo watch e modo set_watch.

O modo *watch* é o responsável pelas funções normais de relógio, ou seja, aquelas de contagem de horas, minutos e segundos, além de mostrar a data corrente e o dia da semana. Quando a aplicação encontra-se neste modo, o mostrador apresenta as informações descritas anteriormente.

O botão **ul** ao ser pressionado neste módulo alterna o modo para *set_watch*. O botão **ll** faz a alternância para o modulo *stop_watch*. O botão **lr** muda o modo de apresentação das horas entre 24h e AM/PM. O botão **ur** é ignorado neste modo (TOSCANI, 1993).

No modo *set_watch* as mesmas informações aparecem, porém este é o modo responsável pelo ajuste das informações do display. A única diferença de interface em relação ao modo *watch* é que a posição a ser acertada pelo usuário é realçada com uma cor que se diferencia das cores padrões do display. A posição que será acertada também não sofre alterações do tempo corrente, ou seja, não sofre alterações pela marcação de tempo normal.

Neste modo, o botão **ul** muda de modo, retornando o programa para o modo *watch*, e o botão **ll** produz o avanço da posição a ser acertada. O botão **lr** aplica um comando de incremento na posição que está sendo ajustada. O botão **ur**, assim como no modo anterior, não produz efeito. Neste modo, deve ser observado que o ambiente continua a ser estimulado pelo sinal **s** e executando suas funções do modo *watch*, porém a posição a ser acertada não sofre mudanças ocasionadas pela contagem normal do tempo.

As funções dos botões em seus respectivos modos são apresentadas ao usuário sempre que o cursor do mouse passar sobre os estes (pequenos *hints*).

Esta interface foi implementada no módulo que contém o nome de *relogio*.

A Figura 17 mostra a interface do relógio digital no modo *set_watch*.

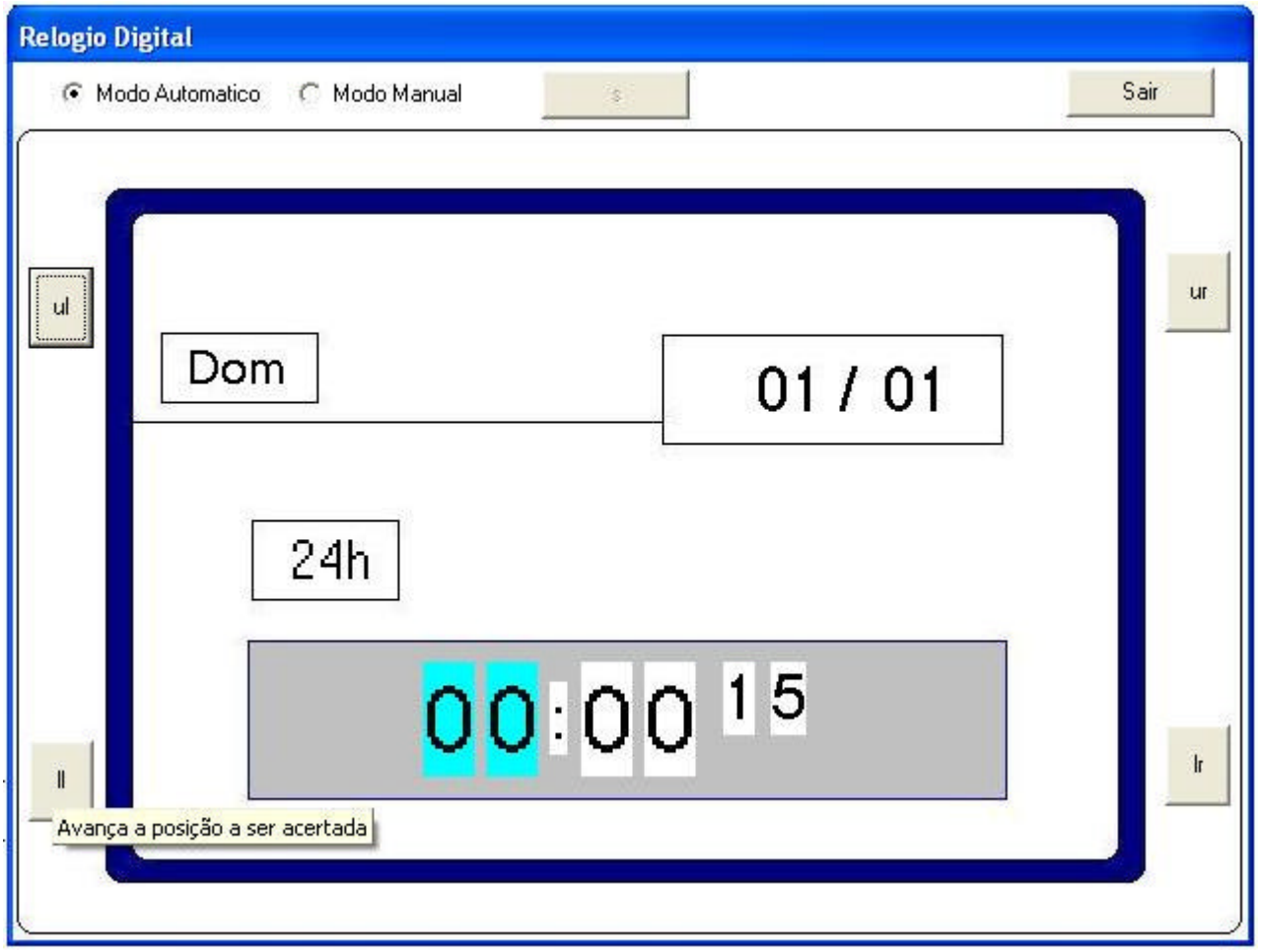

Figura 17 - Relogio Digital

7 Conclusões

O foco deste trabalho era a consolidação do ambiente Exec-RS, com modificações que proporcionassem a interpretação dos mecanismos de exceção da linguagem RS, e a implementação de interfaces específicas para exemplos típicos de programa.

Este trabalho apresentou dificuldades expressivas, por ser uma continuação de dois outros trabalhos, determinando a necessidade de entender a implementação e as estratégias para a realização das alterações necessárias. Além disso, houve uma dificuldade inicial para a ativação do ambiente e execução de programas. Algumas correções foram necessárias para tornar possível a execução de programas RS. Outros erros de implementação foram encontrados durante o processo de implementação do projeto, sendo que a maioria, necessária para o avanço do trabalho, foi resolvida. Alguns detalhes para melhoria da implementação ficam como sugestão para trabalhos futuros.

Para começar este trabalho, foi inicialmente necessário entender muitos conceitos novos relacionados aos sistemas reativos e às características da linguagem RS. Além disso, o funcionamento do código objeto não é convencional, pois baseia-se em um autômato e um conjunto de regras gerados a partir do código fonte. Com o aprofundamento dos conhecimentos sobre os sistemas reativos e a linguagem RS, foi possível um melhor entendimento da implementação realizada anteriormente e com isso detectar os pontos onde deveriam ser feitas as construções necessárias para a consolidação do ambiente, bem como os ajustes que se faziam necessários durante o processo. Juntamente com isso, foi viabilizado o entendimento do protocolo de comunicação entre a interface e o núcleo reativo para a construção das interfaces específicas. As interfaces definidas proporcionam

maior clareza ao usuário e são, naturalmente, mais amigáveis do que a (única) interface genérica disponível anteriormente.

A interface do relógio digital não foi totalmente concluída, pois alguns modos de funcionamento não foram consolidados. Para isso, seria necessária a implementação de mais alguns procedimentos e a definição de mais um sinal de entrada, que deveria ser emitido em intervalos de tempo regulares, visando concluir a simulação de um cronômetro.

Notou-se que o ambiente ainda pode ser melhorado em alguns detalhes da implementação e, com certeza, considerando a sua complexidade, mais erros poderão ser encontrados. Como sugestões de melhorias para trabalhos futuros, podemos citar o procedimento externo *write* que por uma questão de praticidade foi incorporado ao ambiente Exec-RS, porém encontra-se bastante rudimentar. O ambiente também poderia proporcionar mensagens de erros mais claras, além de um tratamento para as mesmas e, até mesmo, um modo de depuração para melhor entendimento do funcionamento dos programas gerados. O sistema de procedimentos externos também pode ter melhorias, pois mesmo que, na forma atual, não sejam encontrados erros, sua eficiência pode ser bastante aumentada.

Com a consolidação deste trabalho, o ambiente Exec-RS está apto para o desenvolvimento de simuladores de sistemas reativos baseados na linguagem RS, o que caracteriza as seguintes contribuições deste trabalho:

- Permitir a execução de códigos da linguagem RS – Plena;
- Ampliar a confiabilidade do sistema;
- Estimular o uso da linguagem RS;
- Validar o protocolo de comunicação entre as interfaces e o executor e demonstrar que interfaces específicas podem ser criadas.

E finalmente, como contribuição pessoal, esse trabalho proporcionou a oportunidade de desenvolver estudos avançados sobre sistemas reativos e um aperfeiçoamento na utilização da linguagem Object Pascal, além de utilizar conhecimentos adquiridos durante o curso de Ciência da Computação, como estrutura de dados, engenharia de software, algoritmos entre tantos outros.

Bibliografia

ARNOLD, Gustavo Vasconcelos. **Uma extensão da linguagem RS para o desenvolvimento de sistemas reativos baseados na arquitetura de subsenção**. Porto Alegre, Brasil: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1998. 95p.

CANTÚ, Marco. **Dominando o Delphi 5 - A bíblia**. Makron Books, 2000. 860p.

COHEN, Jacques. **A View of the Origins and Development of Prolog**. In: *Communications of the ACM*, 31(1): 26-36 (1988). ACM Press.

COLMERAUER, Alain; ROUSSEL, Philippe. **The birth of Prolog**. In: *The 2nd ACM SIGPLAN conference on History of programming Languages*, Cambridge, Massachusetts, USA, 1993. ACM Press.

FARINES, Jean-Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva de. **Sistemas de Tempo Real**. Florianópolis, Brasil: Universidade Federal de Santa Catarina, 2000. 193p.

GIORGI, Ulisses Ponticelli. **A distribuição da linguagem RS**. Porto Alegre, Brasil: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1998. 74p.

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.

LIBRELOTTO, Giovani Rubert. **Um Compilador para a linguagem RS distribuída**. Porto Alegre, Brasil: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2001. 89p.

MATTOS, Julio Carlos B. de. **Geração de código no projeto de sistemas reativos a partir da linguagem RS**. Porto Alegre, Brasil: UFRGS, Dissertação de Mestrado. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2000. 87p.

OLIVEIRA, Alexandre. EXEC-RS - **Um Ambiente de Execução de Programas reativos RS: Incorporação de Recursos Complementares da Linguagem.** Pelotas, Brasil: Universidade Federal de Pelotas, 2005. 66p.

TOSCANI, Simão Sirineo. **RS: Uma Linguagem para Programação de Núcleos Reactivos.** Lisboa, Portugal: Universidade Nova de Lisboa, Dissertação de Doutorado. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1993. 156p.

ZSCHORNACK, Felipe. **Um Ambiente de Execução para a Linguagem RS**. Pelotas, Brasil: Monografia. Instituto de Física e Matemática, Universidade Federal de Pelotas, 2003. 74p.

Apêndice A – Código Objeto do Relógio Digital

O autômato e as regras do relógio digital são vistos a seguir.

```
AUTOMATON relogio
init -    [1, 19, 28, 41, 52, *, go_to(1)]
 1   ul   [2, *, 23, *, 54, *, go_to(2)]
 1   ll   [3, *, 56, *, go_to(3)]
 1   lr   [4, *, 21, 43, *, 42, 53, *, [49-1, *, go_to(4)], [49-2, *,
go_to(1)]]
 1   ur   [18, *, go_to(1)]
 1   s    [20, *, 42, 53, *, [49-1, *, go_to(4)], [49-2, *, go_to(1)]]
 2   ul   [5, *, 27, *, 55, *, go_to(1)]
 2   ll   [6, *, 26, *, 54, 55, *, go_to(2)]
 2   lr   [7, *, 25, *, 42, 53, *, [49-1, *, go_to(5)], [49-2, *,
go_to(2)]]
 2   ur   [18, *, go_to(2)]
 2   s    [24, *, 42, 53, *, [49-1, *, go_to(5)], [49-2, *, go_to(2)]]
 3   ll   [9, *, 60, *, go_to(39)]
 3   lr   [10, *, 29, *, go_to(11)]
 3   ur   [8, 18, *, 37, *, 30, *, 39, *, 59, *, go_to(3)]
 3   s    [20, *, 42, 57, *, [49-1, *, go_to(6)], [49-2, *, go_to(3)]]
 4   ul   [2, *, 23, *, 54, *, go_to(5)]
 4   ll   [3, *, 56, *, go_to(6)]
 4   lr   [4, *, 21, 43, *, 42, 53, *, go_to(4)]
 4   ur   [18, *, 51, *, go_to(1)]
 4   s    [20, *,[50-1, *, 42, 53, *, go_to(4)], [50-2, *, 42, 53, *, [49-
1, *, go_to(4)], [49-2, *, go_to(1)]]]
 5   ul   [5, *, 27, *, 55, *, go_to(4)]
 5   ll   [6, *, 26, *, 54, 55, *, go_to(5)]
 5   lr   [7, *, 25, *, 42, 53, *, go_to(5)]
 5   ur   [18, *, 51, *, go_to(2)]
 5   s    [24, *, [50-1, *, 42, 53, *, go_to(5)], [50-2, *, 42, 53, *, [49-
1, *, go_to(5)], [49-2, *, go_to(2)]]]
 6   ll   [9, *, 60, *, go_to(7)]
 6   lr   [10, *, 29, *, go_to(8)]
 6   ur   [8, 18, *, 37, 51, *, 30, *, 39, *, 59, *, go_to(3)]
 6   s    [20, *, [50-1, *, 42, 57, *, go_to(6)], [50-2, *, 42, 57, *, [49-
1, *, go_to(6)], [49-2, *, go_to(3)]]]
 7   ul   [11, *, 45, *, 62, *, go_to(38)]
 7   ll   [12, *, 58, 64, *, go_to(4)]
 7   lr   [13, *, 22, *, go_to(7)]
 7   ur   [14, 18, *, 44, 51, *, go_to(39)]
 7   s    [20, *, [50-1, *, 42, 57, *, go_to(7)], [50-2, *, 42, 57, *, [49-
1, *, go_to(7)], [49-2, *, go_to(39)]]]
 8   ll   [9, *, 60, *, go_to(9)]
```

```
 8   lr   [10, *, 34, *, go_to(6)]
 8   ur   [8, 18, *, 37, 51, *, 35, *, 38, *, go_to(10)]
 8   hs   [31, *, 33, *, 36, *, 59, *, go_to(8)]
 8   s    [20, 32, *, [50-1, *, 33, 42, 57, *, 36, *, 59, *, go_to(8)],
[50-2, *, 33, 42, 57, *, 36, *, [49-1, *, 59, *, go_to(8)], [49-2, *, 59,
*, go_to(11)]]]
 9   ul   [11, *, 45, *, 62, *, go_to(37)]
 9   ll   [12, *, 58, 64, *, go_to(16)]
 9   lr   [13, *, 22, *, go_to(9)]
 9   ur   [14, 18, *, 44, 51, *, go_to(12)]
 9   hs   [31, *, 33, *, 36, *, go_to(9)]
 9   s    [20, 32, *, [50-1, *, 33, 42, 57, *, 36, *, go_to(9)], [50-2, *,
33, 42, 57, * , 36, *, [49-1, *, go_to(9)], [49-2, *, go_to(12)]]]
10   ll   [9, *, 60, *, go_to(18)]
10   lr   [10, *, 34, *, go_to(19)]
10   ur   [8, 18, *, 40, *, go_to(11)]
10   hs   [31, *, 33, *, go_to(10)]
10   s    [20, 32, *, 33, 42, 57, *, 36, *, [49-1, *, go_to(20)], [49-2, *,
go_to(10)]]
11   ll   [9, *, 60, *, go_to(12)]
11   lr   [10, *, 34, *, go_to(3)]
11   ur   [8, 18, *, 37, *, 35, *, 38, *, go_to(10)]
11   hs   [31, *, 33, *, 36, *, 59, *, go_to(11)]
11   s    [20, 32, *, 33, 42, 57, *, 36, *, [49-1, *, go_to(8)], [49-2, *,
go_to(11)]]
12   ul   [11, *, 45, *, 62, *, go_to(13)]
12   ll   [12, *, 58, 64, *, go_to(14)]
12   lr   [13, *, 22, *, go_to(12)]
12   ur   [14, 18, *, 44, * ,go_to(12)]
12   hs   [31, *, 33, *, 36, *, go_to(12)]
12   s    [20, 32, *, 33, 42, 57, *, 36, *, [49-1, *, go_to(9)], [49-2, *,
go_to(12)]]
13   ul   [15, *, 48, *, 63, *, go_to(12)]
13   ll   [16, *, 47, *, 62, 63, *, go_to(13)]
13   lr   [17, *, 46, *, 61, *, go_to(13)]
13   ur   [18, *, go_to(13)]
13   hs   [31, *, 33, *, 36, *, go_to(13)]
13   s    [20, 32, *, 33, 57, *, 36, *, go_to(13)]
14   ul   [2, *, 23, *, 54, *, go_to(15)]
14   ll   [3, *, 56, *, go_to(11)]
14   lr   [4, *, 21, 43, *, 42, 53, *, [49-1, *, go_to(16)], [49-2, *,
go_to(14)]]
14   ur   [18, *, go_to(14)]
14   hs   [31, *, 33, *, 36, *, go_to(14)]
14   s    [20, 32, *, 33, 42, 53, *, 36, *, [49-1, *, go_to(16)], [49-2, *,
go_to(14)]]
15   ul   [5, *, 27, *, 55, *, go_to(14)]
15   ll   [6, *, 26, *, 54, 55, *, go_to(15)]
15   lr   [7, *, 25, *, 42, 53, *, [49-1, *, go_to(17)], [49-2, *,
go_to(15)]]
15   ur   [18, *, go_to(15)]
15   hs   [31, *, 33, *, 36, *, go_to(15)]
15   s    [24, 32, *, 33, 42, 53, *, 36, *, [49-1, *, go_to(17)], [49-2, *,
go_to(15)]]
16   ul   [2, *, 23, *, 54, *, go_to(17)]
16   ll   [3, *, 56, *, go_to(8)]
16   lr   [4, *, 21, 43, *, 42, 53, *, go_to(16)]
16   ur   [18, *, 51, *, go_to(14)]
16   hs   [31, *, 33, *, 36, *, go_to(16)]
16   s    [20, 32, *, [50-1, *, 33, 42, 53, *, 36, *, go_to(16)], [50-2, *,
33, 42, 53, *, 36, *, [49-1, *, go_to(16)], [49-2, *, go_to(14)]]]
```

```
17   ul   [5, *, 27, *, 55, *, go_to(16)]
17   ll   [6, *, 26, *, 54, 55, *, go_to(17)]
17   lr   [7, *, 25, *, 42, 53, *, go_to(17)]
17   ur   [18, *, 51, *, go_to(15)]
17   hs   [31, *, 33, *, 36, *, go_to(17)]
17   s    [24, 32, *, [50-1, *, 33, 42, 53, *, 36, *, go_to(17)], [50-2, *,
33, 42, 53, *, 36, *, [49-1, *, go_to(17)], [49-2, *, go_to(15)]]]
18   ul   [11, *, 45, *, 62, *, go_to(36)]
18   ll   [12, *, 58, 64, *, go_to(34)]
18   lr   [13, *, 22, *, go_to(18)]
18   ur   [14, 18, *, 44, *, go_to(18)]
18   hs   [31, *, 33, *, go_to(18)]
18   s    [20, 32, *, 33, 42, 57, *, [49-1, *, go_to(21)], [49-2, *,
go_to(18)]]
19   ll   [9, *, 60, *, go_to(26)]
19   lr   [10, *, 29, *, go_to(10)]
19   ur   [8, 18, *, 40, *, go_to(3)]
19   s    [20, *, 42, 57, *, [49-1, *, go_to(22)], [49-2, *, go_to(19)]]
20   ll   [9, *, 60, *, go_to(21)]
20   lr   [10, *, 34, *, go_to(22)]
20   ur   [8, 18, *, 40, 51, *, go_to(11)]
20   hs   [31, *, 33, *, go_to(20)]
20   s    [20, 32, *, [50-1, *, 33, 42, 57, *, go_to(20)], [50-2, *, 33,
42, 57, *, [49-1, *, go_to(20)], [49-2, *, go_to(10)]]]
21   ul   [11, *, 45, *, 62, *, go_to(31)]
21   ll   [12, *, 58, 64, *, go_to(32)]
21   lr   [13, *, 22, *, go_to(21)]
21   ur   [14, 18, *, 44, 51, *, go_to(18)]
21   hs   [31, *, 33, *, go_to(21)]
21   s    [20, 32, *, [50-1, *, 33, 42, 57, *, go_to(21)], [50-2, *, 33,
42, 57, *, [49-1, *, go_to(21)], [49-2, *, go_to(18)]]]
22   ll   [9, *, 60, *, go_to(23)]
22   lr   [10, *, 29, *, go_to(20)]
22   ur   [8, 18, *, 40, 51, *, go_to(3)]
22   s    [20, *, [50-1, *, 42, 57, *, go_to(22)], [50-2, *, 42, 57, *,
[49-1, *, go_to(22)], [49-2, *, go_to(19)]]]
23   ul   [11, *, 45, *, 62, *, go_to(24)]
23   ll   [12, *, 58, 64, *, go_to(25)]
23   lr   [13, *, 22, *, go_to(23)]
23   ur   [14, 18, *, 44, 51, *, go_to(26)]
23   s    [20, *, [50-1, *, 42, 57, *, go_to(23)], [50-2, *, 42, 57, *,
[49-1, *, go_to(23)], [49-2, *, go_to(26)]]]
24   ul   [15, *, 48, *, 63, *, go_to(23)]
24   ll   [16, *, 47, *, 62, 63, *, go_to(24)]
24   lr   [17, *, 46, *, 61, *, go_to(24)]
24   ur   [18, *, 51, *, go_to(27)]
24   s    [20, *, [50-1, *, 57, *, go_to(24)], [50-2, *, 57, *, go_to(27)]]
25   ul   [2, *, 23, *, 54, *, go_to(30)]
25   ll   [3, *, 56, *, go_to(22)]
25   lr   [4, *, 21, 43, *, 42, 53, *, go_to(25)]
25   ur   [18, *, 51, *, go_to(28)]
25   s    [20, *, [50-1, *, 42, 53, *, go_to(35)], [50-2, *, 42, 53, *,
[49-1, *, go_to(25)], [49-2, *, go_to(28)]]]
26   ul   [11, *, 45, *, 62, *, go_to(27)]
26   ll   [12, *, 58, 64, *, go_to(28)]
26   lr   [13, *, 22, *, go_to(26)]
26   ur   [14, 18, *, 44, *, go_to(26)]
26   s    [20, *, 42, 57, *, [49-1, *, go_to(23)], [49-2, *, go_to(26)]]
27   ul   [15, *, 48, *, 63, *, go_to(26)]
27   ll   [16, *, 47, *, 62, 63, *, go_to(27)]
27   lr   [17, *, 46, *, 61, *, go_to(27)]
```

```
27   ur   [18, *, go_to(27)]
27   s    [20, *, 57, *, go_to(27)]
28   ul   [2, *, 23, *, 54, *, go_to(29)]
28   ll   [3, *, 56, *, go_to(19)]
28   lr   [4, *, 21, 43, *, 42, 53, *, [49-1, *, go_to(25)], [49-2, *,
go_to(28)]]
28   ur   [18, *, go_to(28)]
28   s    [20, *, 42, 53, *, [49-1, *, go_to(25)], [49-2, *, go_to(28)]]
29   ul   [5, *, 27, *, 55, *, go_to(28)]
29   ll   [6, *, 26, *, 54, 55, *, go_to(29)]
29   lr   [7, *, 25, *, 42, 53, *, [49-1, *, go_to(30)], [49-2, *,
go_to(29)]]
29   ur   [18, *, go_to(29)]
29   s    [24, *, 42, 53, *, [49-1, *, go_to(30)], [49-2, *, go_to(29)]]
30   ul   [5, *, 27, *, 55, *, go_to(25)]
30   ll   [6, *, 26, *, 54, 55, *, go_to(30)]
30   lr   [7, *, 25, *, 42, 53, *, go_to(30)]
30   ur   [18, *, 51, *, go_to(29)]
30   s    [24, *, [50-1, *, 42, 53, *, go_to(30)], [50-2, *, 42, 53, *,
[49-1, *, go_to(30)], [49-2, *, go_to(29)]]]
31   ul   [15, *, 48, *, 63, *, go_to(21)]
31   ll   [16, *, 47, *, 62, 63, *, go_to(31)]
31   lr   [17, *, 46, *, 61, *, go_to(31)]
31   ur   [18, *, 51, *, go_to(36)]
31   hs   [31, *, 33, *, go_to(31)]
31   s    [20, 32, *, [50-1, *, 33, 57, *, go_to(31)], [50-2, *, 33, 57, *,
go_to(36)]]
32   ul   [2, *, 23, *, 54, *, go_to(33)]
32   ll   [3, *, 56, *, go_to(20)]
32   lr   [4, *, 21, 43, *, 42, 53, *, go_to(32)]
32   ur   [18, *, 51, *, go_to(34)]
32   hs   [31, *, 33, *, go_to(32)]
32   s    [20, 32, *, [50-1, *, 33, 42, 53, *, go_to(32)], [50-2, *, 33,
42, 53, *, [49-1, *, go_to(32)], [49-2, *, go_to(34)]]]
33   ul   [5, *, 27, *, 55, *, go_to(32)]
33   ll   [6, *, 26, *, 54, 55, *, go_to(33)]
33   lr   [7, *, 25, *, 42, 53, *, go_to(33)]
33   ur   [18, *, 51, *, go_to(35)]
33   hs   [31, *, 33, *, go_to(33)]
33   s    [24, 32, *, [50-1, *, 33, 42, 53, *, go_to(33)], [50-2, *, 33,
42, 53, *, [49-1, *, go_to(33)], [49-2, *, go_to(35)]]]
34   ul   [2, *, 23, *, 54, *, go_to(35)]
34   ll   [3, *, 56, *, go_to(10)]
34   lr   [4, *, 21, 43, *, 42, 53, *, [49-1, *, go_to(32)], [49-2, *,
go_to(34)]]
34   ur   [18, *, go_to(34)]
34   hs   [31, *, 33, *, go_to(34)]
34   s    [20, 32, *, 33, 42, 53, *, [49-1, *, go_to(32)], [49-2, *,
go_to(34)]]
35   ul   [5, *, 27, *, 55, *, go_to(34)]
35   ll   [6, *, 26, *, 54, 55, *, go_to(35)]
35   lr   [7, *, 25, *, 42, 53, *, [49-1, *, go_to(33)], [49-2, *,
go_to(35)]]
35   ur   [18, *, go_to(35)]
35   hs   [31, *, 33, *, go_to(35)]
35   s    [24, 32, *, 33, 42, 53, *, [49-1, *, go_to(33)], [49-2, *,
go_to(35)]]
36   ul   [15, *, 48, *, 63, *, go_to(18)]
36   ll   [16, *, 47, *, 62, 63, *, go_to(36)]
36   lr   [17, *, 46, *, 61, *, go_to(36)]
36   ur   [18, *, go_to(36)]
```

```
36   hs   [31, *, 33, *, go_to(36)]
36   s    [20, 32, *, 33, 57, *, go_to(36)]
37   ul   [15, *, 48, *, 63, *, go_to(9)]
37   ll   [16, *, 47, *, 62, 63, *, go_to(37)]
37   lr   [17, *, 46, *, 61, *, go_to(37)]
37   ur   [18, *, 51, *, go_to(13)]
37   hs   [31, *, 33, *, 36, *, go_to(37)]
37   s    [20, 32, *, [50-1, *, 33, 57, *, 36, *, go_to(37)], [50-2, *, 33,
57, *, 36, *, go_to(13)]]
38   ul   [15, *, 48, *, 63, *, go_to(7)]
38   ll   [16, *, 47, *, 62, 63, *, go_to(38)]
38   lr   [17, *, 46, *, 61, *, go_to(38)]
38   ur   [18, *, 51, *, go_to(40)]
38   s    [20, *, [50-1, *, 57, *, go_to(38)], [50-2, *, 57, *, go_to(40)]]
39   ul   [11, *, 45, *, 62, *, go_to(40)]
39   ll   [12, *, 58, 64, *, go_to(1)]
39   lr   [13, *, 22, *, go_to(39)]
39   ur   [14, 18, *, 44, *, go_to(39)]
39   s    [20, *, 42, 57, *, [49-1, *, go_to(7)], [49-2, *, go_to(39)]]
40   ul   [15, *, 48, *, 63, *, go_to(39)]
40   ll   [16, *, 47, *, 62, 63, *, go_to(40)]
40   lr   [17, *, 46, *, 61, *, go_to(40)]
40   ur   [18, *, go_to(40)]
40   s    [20, *, 57, *, go_to(40)]

RULES FOR relogio
Module button:
  1.   [] ===> []
  2.   [] ===> [emit(enter_set_W_M_com)]
  3.   [] ===> [emit(stpw_M_com)]
  4.   [] ===> [emit(toggle_24H_M_com)]
  5.   [] ===> [emit(exit_set_W_M_com)]
  6.   [] ===> [emit(next_W_T_P_com)]
  7.   [] ===> [emit(set_W_com)]
  8.   [] ===> [emit(lap_com)]
  9.   [] ===> [emit(alarm_M_com)]
 10.   [] ===> [emit(start_stop_com)]
 11.   [] ===> [emit(enter_set_A_M_com)]
 12.   [] ===> [emit(watch_M_com)]
 13.   [] ===> [emit(toggle_chime_com)]
 14.   [] ===> [emit(toggle_A_com)]
 15.   [] ===> [emit(exit_set_A_M_com)]
 16.   [] ===> [emit(next_A_T_P_com)]
 17.   [] ===> [emit(set_A_com)]
 18.   [] ===> [emit(stop_A_beep_com)]
Module watch:
 19.   [] ===> [set_rec(watch_t(0)), chime_s:=0, emit(chime_S(0))]
 20.   [] ===> [get_rec(watch_T(A)), incr_W_T(A,B), set_rec(watch_T(B)),
emit(watch_Time(B,0)), get_watch_beep_val(B,C), emit(beep(C))]
 21.   [] ===> [get_rec(watch_T(A)), toggle_24H_M_in_W_T(A,B),
set_rec(watch_T(B)), emit(watch_Time(B,0))]
 22.   [] ===> [chime_S := 1 - chime_S, emit(chime_S(chime_S))]
 23.   [] ===> [watch_T_pos := 0, emit(start_E(0))]
 24.   [] ===> [get_rec(watch_T(A)), incr_W_T_in_set_M(A,watch_T_pos,B),
set_rec(watch_T(B)), emit(watch_Time(B,1))]
 25.   [] ===> [get_rec(watch_T(A)), set_W_T(A,watch_T_pos,B),
set_rec(watch_T(B)), emit(watch_Time(B,1))]
 26.   [] ===> [emit(stop_E(watch_T_pos)), watch_T_pos := watch_T_pos + 1,
watch_T_pos:= watch_T_pos mod 6, emit(start_E(watch_T_pos))]
 27.   [] ===> [emit(stop_E(watchT_pos))]
Module stopwatch:
```

```
 28.   [] ===> [emit(stpw_Run_S(off)), emit(stpw_Lap_S(off)),
set_rec(stpw_T(0))]
 29.   [] ===> [emit(stpw_Run_S(on)), emit(beep(1))]
 30.   [] ===> []
 31.   [] ===> []
 32.   [] ===> []
 33.   [] ===> [get_rec(stpw_T(A)), incr_stpw_T(A,B), set_rec(stpw_T(B)),
up(new_stpw_T(B)), get_stpw_beep_val(B,C), emit(beep(C))]
 34.   [] ===> [emit(stpw_Run_S(off)), emit(beep(1))]
 35.   [] ===> []
 36.   [new_stpw_T(A)] ===> [emit(stpw_Time(A))]
 37.   [] ===> []
 38.   [] ===> [emit(stpw_Lap_S(on))]
 39.   [] ===> [emit(stpw_Run_S(off)), emit(stpw_Lap_S(off)),
set_rec(stpw_T(0)), emit(stpw_Time(0))]
 40.   [] ===> [emit(stpw_Lap_S(off))]
Module alarm:
 41.   [] ===> [set_rec(alarm_T(0)), emit(alarm_Time(0)), alarm_S := 0,
emit(alarm_S(0))]
 42.   [watch_Time(A,B)] ===> [get_rec(alarm_T(C)),
compare_A_T_to_W_T(C,A,B,D), up(start_beeping(D))]
 43.   [] ===> [get_rec(alarm_T(A)), toggle_24H_M_in_A_T(A,B),
set_rec(alarm_T(B)), emit(alarm_Time(B))]
 44.   [] ===> [alarm_S := 1 - alarm_S, emit(alarm_S(alarm_S))]
 45.   [] ===> [alarm_T_pos := 0, emit(start_E(0))]
 46.   [] ===> [get_rec(alarm_T(A)), set_A_T(A, alarm_T_pos, B),
set_rec(alarm_T(B)), emit(alarm_Time(B))]
 47.   [] ===> [emit(stop_E(alarm_T_pos)), alarm_T_pos := 1 - alarm_T_pos,
emit(start_E(alarm_T_pos))]
 48.   [] ===> [emit(stop_E(alarm_T_pos)), alarm_S := 1,
emit(alarm_S(alarm_S))]
 49.   Case:
 49-1. [start_beeping(A)] {A=1} ---> [alarm_count:=30]
 49-2. [start_beeping(A)] {else} ---> []
 50.   Case:
 50-1. [] {alarm_count>0} ---> [alarm_count:=alarm_count - 1,
emit(beep(4))]
 50-2. [] {else} ---> []
 51.   [] ===> []
Module display:
 52.   [] ===> []
 53.   [watch_Time(A,B)] ===> [watch_T_to_main_D(A,C), emit(main_D(C)),
watch_Date_to_mini_D(A,D), emit(mini_D(D)), watch_Day_to_alph_D(A,E),
emit(alph_D(E))]
 54.   [start_E(A)] ===> [watch_D_P(A,B), emit(start_E_D(B))]
 55.   [stop_E(A)] ===> [watch_D_P(A,B), emit(stop_E_D(B))]
 56.   [] ===> [emit(alph_D(A))]
 57.   [watch_Time(A,B)] ===> [watch_T_to_mini_D(A,C), emit(mini_D(C))]
 58.   [] ===> []
 59.   [stpw_Time(A)] ===> [stpw_T_to_main_D(A,B), emit(main_D(B))]
 60.   [] ===> [emit(alph_D(A))]
 61.   [alarm_Time(A)] ===> [alarm_T_to_main_D(A,B), emit(main_D(B))]
 62.   [start_E(A)] ===> [alarm_D_P(A,B), emit(start_E_D(B))]
 63.   [stop_E(A)] ===> [alarm_D_P(A,B), emit(stop_E_D(B))]
 64.   [] ===> []
```