Capítulo 4

O Modelo de Programação Síncrona para os Sistemas de Tempo Real

4.1 Introdução

Os *Sistemas de Tempo Real* são sistemas que reagem, gerando respostas, à estímulos de entrada vindos de seus ambientes. Estas características os colocam como *Sistemas Reativos* especiais que estão submetidos a restrições temporais em suas reações. Neste tipo de sistemas, devem se garantir não somente a correção lógica ("*correctness*") mas também a correção temporal ("*timeliness*").

O modelo de programação síncrona parte do principio que o ambiente não interfere com o sistema (ou o programa) durante os processamentos das reações. Na recepção do evento de entrada, após um eventual cálculo, a resposta é considerada emitida simultaneamente à entrada, o que caracteriza uma reação como instantânea. O modelo é dito *síncrono* porque as saídas do sistema podem ser vistas como sincronizadas com as suas entradas. A principal conseqüência desta hipótese – a *Hipótese Síncrona* – é uma simplificação conceptual que facilita a modelagem e a análise formal das propriedades do sistema.

A hipótese síncrona, que forma a base conceitual deste estilo de programação, no sentido da reação instantânea, parte do pressuposto que existe uma máquina suficientemente rápida para executar o processamento correspondente à reação em tempos não significativos. O entendimento desta hipótese, em um sentido mais prático, é que a duração do processamento referente a reação, se comparado aos tempos relacionados com o ambiente externo, é desprezível. O ambiente externo não evolui durante esse processamento. A hipótese síncrona também define que os eventos ocorridos no sistema sejam percebidos instantaneamente em diferentes partes do sistema.

O modelo de programação síncrono é natural do ponto de vista do programador por facilitar a construção e a compreensão de programas e por lhe permitir a verificação destes. Ao separar a lógica do comportamento de um sistema das características de sua implementação, esse estilo de programação facilita a programação do mesmo.

Uma das características mais importantes encontrada nos modelos síncronos é a rejeição do não determinismo. Um sistema é visto como determinista se a mesma

seqüência de entradas produz sempre a mesma seqüência de saídas e não determinista em caso contrário. É desejável e quase sempre mandatário que os sistemas reativos se comportem de forma determinista, com as suas saídas dependendo unicamente das entradas vindas do ambiente e de exigências temporais; o comportamento esperado no controle de um avião ou de outro veículo, por exemplo ilustram esta necessidade. Além do mais o estudo do comportamento de sistemas não deterministas é mais complexo que o de sistemas deterministas por apresentar situações de erro que podem não ser reproduzíveis e por tratar com grande número de estados. Consequentemente é aconselhável tratar sistemas intrinsecamente deterministas como os sistemas reativos por modelos e linguagens deterministas e reservar abordagens não deterministas as aplicações que o necessitam como é o caso de aplicações nas quais não há como garantir comportamentos repetitivos nem garantir tempos de resposta; Internet é um exemplo claro deste comportamento.

As linguagens síncronas que apresentam características de concorrência e de determinismo e permitem ter o controle dos tempos de respostas são bem adaptadas para a programação dos Sistemas Reativos e dos Sistemas de Tempo Real. Ferramentas automáticas que possibilitam a verificação da correção lógica e temporal desses sistemas são associadas à estas linguagens. A hipótese de sincronismo permite ainda que programas escritos numa linguagem síncrona sejam compilados em autômatos eficientes e depois facilmente implementados em linguagens de programação clássicas.

Os Sistemas Reativos e os Sistemas de Tempo Real incluem em doses diferentes segundo as aplicações, atividades de manuseio de dados e atividades de controle. Quando as atividades de manuseio de dados são importantes e complexas enquanto as de controle são reduzidas como em aplicações de processamento de sinal, as técnicas de especificação e de programação mais apropriadas seguem um estilo orientado a fluxo de dados como nas linguagens síncronas declarativas Lustre [HCR91] e Signal [LLG91]. Nestas linguagens, a reação gera saídas a partir da avaliação de um conjunto de equações que as definem em função das entradas atuais e das entradas previas (armazenadas).

Quando predominam as atividades de controle e que o manuseio de dados é simples, como é o caso em aplicações de controle de processos, sistemas embutidos, supervisão de sistemas, protocolos de comunicação, interfaces homem-máquina, drivers de periféricos, entre outras, é mais apropriada adotar um estilo orientado ao fluxo de controle como nas linguagens síncronas imperativas como Esterel [BoS91] ou nos autômatos hierárquicos como Statecharts [Har87]. Nestas linguagens, cada reação corresponde a passagem de uma situação em termos de controle à uma nova situação.

Grandes aplicações podem ter partes mais orientadas ao manuseio de dados e outras ao controle; no momento atual, não existe ainda unificação entre os dois estilos ao nível de linguagem de programação, apesar de existir atividades de pesquisa nesta área. Neste livro, adotaremos a linguagem Esterel como ferramenta para a programação de aplicações tempo real dentro do modelo de programação síncrona, por ela ser imperativa, ter uma sintaxe e semântica de fácil aprendizagem e por ter disponível um ambiente automático de especificação, validação e implementação.

4.2 Princípios Básicos do Modelo de Programação da Linguagem Esterel

A linguagem Esterel é uma linguagem síncrona desenvolvida a partir de 1982 conjuntamente por dois laboratórios franceses do INRIA e da ENSMP. A necessidade de atender simultaneamente concorrência e determinismo é a base do modelo de programação síncrono da linguagem Esterel. Os princípios básicos deste modelo são os seguintes:

- *Reatividade*: O modelo é reativo uma vez que se aplica a sistemas que entram em ação reagindo à presença de estímulos vindos do ambiente em instantes discretos. Cada reação, portanto, está associada a um instante preciso; o conjunto destes instantes caracteriza a vida do sistema reativo.

- *Sincronismo*: As reações sendo instantâneas, entradas e saídas se apresentam sincronizadas – é a base da hipótese síncrona. As reações são atômicas, o modelo síncrono não permite uma nova ativação do sistema enquanto o mesmo estiver reagindo ao estímulo atual. Portanto, não há concorrência entre as reações, eliminando assim uma fonte de não determinismo que corresponderia ao entrelaçamento ("*interleaving*") de execuções concorrentes.

- *Difusão instantânea*: A comunicação entre componentes é sempre realizada por um mecanismo de difusão instantânea ("*broadcasting*"). Um sinal emitido é visto no mesmo instante da sua emissão por todos seus receptores. A difusão é limitada aos instantes de reação: um sinal emitido num instante é visto como presente em todos os receptores neste mesmo instante; pode haver entretanto varias emissões e recepções de sinal em seqüência num mesmo instante.

- *Determinismo*: Contrariamente as abordagens assíncronas, onde a concorrência leva ao não determinismo, o modelo faz conviver concorrência e determinismo. O modelo síncrono é determinista, o que tem como resultado a simplificação das programações reativas e a capacidade de reproduzir seus comportamentos, simplificando testes e verificações destas.

Na abordagem síncrona, o tempo é considerado como uma entidade multiforme sendo visto como um evento externo entre outros, de características diversas do tempo físico; a noção de tempo físico é na verdade substituída pela noção de ordem e de simultaneidade entre eventos. Essa visão de tempo multiforme e a não ocorrência de "*interleaving*" facilitam em muito o entendimento e a análise dos sistemas de tempo real.

Técnicas de compilação geram a partir destes modelos síncronos – geralmente descritos na forma de linguagens – autômatos de estado finito deterministas, nos quais

o paralelismo e a comunicação expressos no formalismo inicial desaparecem; em decorrência disto, estes autômatos apresentam um grau elevado de eficiência e previsibilidade temporal. A estas características, são adicionados a simplicidade do autômato resultante em termos do número de estados e o uso de técnicas de verificação ou de prova que são facilitadas. Estes autômatos são também facilmente implementados em qualquer linguagem clássica (C, Java, etc.) ou mesmo, em lógica de circuitos seqüências booleanos.

4.3 O Estilo de Programação da Linguagem Esterel

4.3.1 Programando num estilo imperativo

Vamos a seguir a partir de um pequeno exemplo mostrar a facilidade de programação síncrona, usando o estilo imperativo da linguagem Esterel e introduzir algumas das construções de base que o caracterizam.

Considera a especificação informal:

"*Emite uma saída O, tão logo que as duas entradas A e B tenham ocorrido. Reinicialize este comportamento a cada vez que ocorrer uma entrada R*".

O comportamento descrito acima pode ser representado por vários formalismos – entre estes, por um autômato. Na figura 4.1, é apresentado o autômato [Ber99] que representa a especificação anterior com 4 estados, 8 transições; as entradas e saídas

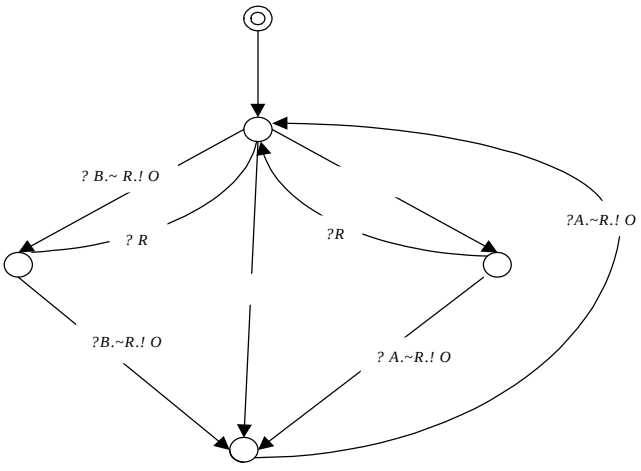

Figura 4.1: Autômato para a especificação ABRO

aparecem várias vezes (3 vezes para A, B e O e 8 vezes para R) neste autômato, tornando a representação complexa o que permite concluir na inadequação de realizar uma especificação direta na forma de um autômato. Além do mais, se o número de entradas cresce, aumenta consequentemente a complexidade na representação por causa da explosão exponencial do autômato.

A linguagem imperativa síncrona Esterel permite ao usuário escrever de forma simples e natural, sem repetição a especificação anterior no módulo ABRO:

```
module ABRO:
input A, B, R;
output O;
loop
        [await A || await B];
        emit O
each R
end module
```

Note que na especificação acima, diferente do autômato, ocorrem uma só vez as entradas e saída A, B, R e O. O aumento do número de entradas é também facilmente absorvido por este estilo de programação e o módulo ABCRO conteria apenas esta modificação:

```
loop
        [await A || await B || await C];
        emit O
each R
```

O princípio de base de um bom estilo de programação consiste em escrever cada parte da especificação apenas uma vez, evitando repetições que dificultam o entendimento e a manutenção do programa e podem ser eventuais fontes de erro. As construções da linguagem Esterel são particularmente bem adaptadas para ajudar o usuário a gerar programas para sistemas reativos que seguem este estilo de programação.

O código anterior contém algumas construções básicas da linguagem Esterel:

- *O atraso*: a construção temporal "**await**" significa a espera por um evento. Quando iniciada, corresponde a uma pausa até o evento ocorrer, instante no qual conclui a operação: "**await A**" espera o sinal **A** e termina quando este ocorre.

- *A emissão de sinal*: A emissão instantânea de sinal é realizada pela construção "**emit ...**", No exemplo acima "**emit O**" corresponde a emissão instantânea do

sinal **O**, tão logo a última entrada **A** ou **B** seja recebida. Não pode ocorrer mais de um "**emit**" por instante (a regra correspondente será vista mais tarde).

- *Seqüência*: "**p;q**" transfere imediatamente o controle a **q**, quando **p** termina.

- *Concorrência*: O operador de paralelismo "**||**" define as construções separadas pelo operador em paralelismo síncrono. A menos da intervenção de algum mecanismo de preempção ou de exceção, a construção termina quando todos seus ramos terminaram. Neste exemplo, "**await A || await B**" termina instantaneamente desde que as duas componentes concluam com as duas entradas **A** e **B** sendo recebidas.

- *aborto ou preempção*: Na construção " **loop p each R**", o corpo **p** é imediatamente inicializado e executa repetidamente até o instante de ocorrência de **R** no qual **p** é abortado e imediatamente reinicializado; esta construção é dita de *aborto ou preempção forte* pois o evento **R** é prioritário sobre o corpo em execução. No exemplo do módulo ABRO se **A**, **B** e **R** ocorrem simultaneamente, **O** não será emitido.

O estado de um sinal envolvido numa reação é *presente* ou *ausente*. O ambiente fornece o estado dos sinais de entrada, e a execução da instrução **emit** transforma o estado dos outros sinais de *ausente* para *presente*.

A comunicação em Esterel é realizada por difusão instantânea. Na entrada, a difusão instantânea é implícita para as instruções concorrentes. A difusão instantânea vale para todos os sinais e permite que processos interessados num sinal emitido (**emit O)** esperem simplesmente o mesmo através de uma instrução **await** por exemplo, sem ter que revelar suas existências ao módulo emissor e independentemente do número de processos receptores.

Após esta breve introdução ao estilo de programação da linguagem Esterel, vamos apresentar as principais construções que caracterizam este estilo, descrevendo declarações e comportamentos no contexto de pequenos exemplos ilustrativos. Uma apresentação mais detalhada da sintaxe e semântica da linguagem Esterel pode ser encontrada no Anexo C.

4.3.2 Declaração de interface

Seja a especificação de um medidor de velocidade descrito informalmente por

"*Contar o número de centímetros por segundo e difundi-lo a cada segundo como sendo o valor de um sinal* ***Velocidade***".

Os sinais de entrada do módulo medidor de velocidade são gerados a cada centímetro e a cada segundo e são representados por **Centímetro** e **Segundo** que definem cada um, uma unidade de tempo independente, caracterizando desta forma que

o tempo é multiforme no modelo síncrono. Para simplificar o exemplo, supõe-se que esses dois sinais não podem ser simultâneos (hipótese plausível devido ao ambiente de execução); a relação de exclusividade de um sinal se representa por **#** numa declaração **relation**. A difusão do valor da velocidade será feita por um sinal com valor, **Velocidade**, que a cada instante além do seu estado contém um valor com tipo **integer**.

O código do módulo medidor de velocidade em Esterel se escreve como:

```
module Medidor-Velocidade:
input Centímetro, Segundo;
relation Centímetro # Segundo;
output Velocidade : integer;
loop
        var Distancia := 0 : integer in
                abort
                        every Centímetro do
                                Distancia := Distancia + 1
                        end every
                when Segundo do
                        emit Velocidade (Distancia)
                end abort
        end var
end loop
end module
```

O estilo de programação adotado em Esterel prioriza o controle de atividades pelo uso da preempção. Dentro da malha infinita "**loop ... end**", a construção "**abort ... when ... do ... end abort**" interrompe seu corpo quando o sinal **Segundo** ocorre e executa a claúsula de *timeout* que segue o "**do**" que permite emitir o sinal **Velocidade** que contém o ultimo valor **Distancia**. Dentro do corpo do "**abort**" uma malha "**every ... end every**" permite incrementar a variável **Distancia** a cada sinal de entrada **Centímetro**; esta malha "**every**" difere da malha "**loop**" anteriormente apresentada por depender da ocorrência de um sinal (neste exemplo **Centímetro**) para iniciar o seu corpo. Considerando a transmissão do controle no programa e as operações suficientemente rápidas para serem vistas como instantâneas dentro da hipótese de sincronismo, a velocidade é emitida exatamente quando ocorre o *tick* indicando o segundo.

Um sinal com valor (como **Velocidade** neste exemplo) tem um único estado e um único valor a cada reação; contrariamente ao estado, o valor deste permanece igual ao da reação anterior até sofrer mudança. O ambiente determina o valor para os sinais de entrada e as instruções "**emit**" para os sinais de saída. A expressão "**?S**" permite acessar o valor corrente de um sinal **S**. O estado e o valor de um sinal podem ser difundidos instantaneamente.

A construção "**relation ...**" permite representar condições supostamente garantidas

pelo ambiente entre os sinais do tipo "**input**" (e também do tipo "**return**"). Os dois tipos de relação são de incompatibilidade ou exclusão entre os sinais ("**#**") como é o caso neste exemplo e de sincronização entre sinais ("=>"). As relações são úteis para evitar a programação de situações de menor importância e reduzir em conseqüência o tamanho do autômato gerado, facilitando a sua verificação.

Em algumas situações onde é apenas necessário a leitura de um valor a qualquer momento, sem a necessidade de gerar também um evento de entrada, pode se utilizar o sinal de entrada "**sensor**" que tem apenas o valor mas sem a presença da informação de estado. Assim como o sinal com valor, definido anteriormente, o sensor tem o seu valor lido pelo operador "**?**".

4.3.3 Declaração de variáveis

Uma variável é um objeto ao qual podem ser atribuídos valores. Uma variável é local à uma "*thread*" que corresponde ao corpo **p** de um módulo (no exemplo anterior a variável **Distancia** é incrementada a cada sinal de entrada **Centímetro** dentro da malha "**every ... end every**" do corpo do "**abort**"). O valor da variável pode ser fornecido por uma instrução de atribuição instantânea ou passado numa chamada de um procedimento externo instantâneo ou ainda passado pela instrução "**exec**" que permite a execução de tarefas externas de cálculo que tomam tempo (esta instrução será vista futuramente). Contrariamente ao sinal, uma variável pode tomar vários valores sucessivos no mesmo instante.

4.3.4 Os diferentes tipos de preempção

Foi percebida nos exemplos anteriores a importância para o estilo de programação Esterel, da construção de preempção que permite matar o seu corpo quando um evento ocorre ou um prazo se esgota. Foi apresentado até o momento um tipo de preempção chamado forte que é expressado pelas construções "**abort p when S**" e "**loop p each S**". Entretanto é interessante poder fornecer ao usuário outras opções com semânticas diferentes.

O comportamento da construção de preempção forte "**abort p when S**" é o seguinte:

- no começo da execução, **p** é imediatamente inicializado, independentemente da presença ou ausência de sinal **S**;
- se **p** termina antes da ocorrência de **S**, então a instrução "**abort**" termina neste mesmo instante;
- se **S** ocorre antes do término de **p**, então a instrução "**abort**" termina imediatamente e **p** não recebe mais o controle.

Comportamentos diferentes podem ser necessários para disponibilizar ao usuário. A possibilidade de terminação da instrução "**abort**" desde o primeiro instante mesmo sem **p** ter iniciado, é permitida pela construção "**abort p when immediate S**" que introduz a palavra chave "**immediate**".

Para permitir que o corpo **p** receba ainda o controle no instante da preempção, para uma última atividade, utiliza-se uma construção de preempção fraca: "**weak abort p when S**". O comportamento de preempção fraca é também possível através da construção "**weak abort p when immediate S**". Para o exemplo do módulo ABRO citado anteriormente, a utilização da construção "**weak abort**" no lugar de "**loop ... each**" torna possível programar, se desejado, a emissão do sinal de saída **O** mesmo no caso de uma situação de simultaneidade entre os sinais **A**, **B**, **R**. Da mesma forma, o uso de "**weak abort**" no módulo Medidor-Velocidade permite no caso de ser autorizada a ocorrência simultânea dos sinais **Centímetro** e **Segundo (**a relação de exclusividade **#** seria retirada do exemplo anterior), de levar em conta o último incremento da variável **Distancia** antes que a velocidade seja emitida; a utilização da preempção forte "**abort**" neste caso levaria a ignorar o último centímetro na instrução "**every Centímetro**" por ter sido abortada pela ocorrência de **Segundo**.

A construção em Esterel "**every S do p end**" encontrada no módulo Medidor-Velocidade é uma abreviação de:

```
await S;
loop
        abort [p; halt] when S
end loop
```

e permite realizar também uma preempção forte de seu corpo e a seguir ser atrasado (**halt** é uma primitiva que significa "espera para sempre"). Esta construção tem ainda uma forma imediata "**every immediate S do p end**" que permite levar em conta o sinal desde o primeiro instante. A solução do problema da simultaneidade de **Centímetro** e **Segundo** no módulo Medidor-Velocidade pode ser também resolvido usando um "**abort**" forte e um "**every immediate**"; neste caso o centímetro não será contado durante o segundo que esta acabando más será levado em conta no inicio do próximo segundo.

Apesar de ter discutido essas construções apenas em termos de sinal, expressões de sinais – que são na verdade expressões booleanas formadas com operadores "**not**", "**and**", e "**or**" – aplicadas sobre o estado de vários sinais podem ser utilizadas nas construções temporais "**abort**", "**every**".

A linguagem Esterel fornece ainda um outro tipo de preempção mais brando que tem um efeito de suspensão (do tipo ^Z do Unix em contraposição com o tipo ^C mais duro da construção "**abort**"): "**suspend p when <S ou expressão-sinal>**". Quando a construção "**suspend**" inicia, o corpo **p** é imediatamente iniciado. A cada instante, se o sinal **S** esta presente ou a expressão de sinal for "*true*", o corpo **p** se suspende e é

congelado no estado no qual se encontrava no momento da suspensão; em caso contrario, o corpo **p** é executado neste instante. Como nas outras construções de preempção, é necessário usar a palavra chave "**immediate**" na construção "**suspend p when immediate <S ou expressão-sinal>**" para poder testar a eventual presença do sinal **S** ou a expressão de sinal no instante inicial.

4.3.5 Mecanismo de exceção

O mecanismo de exceção da linguagem Esterel é implementado pela construção "**trap T in p end trap**" que permite programar um ponto de saída para o corpo **p**. O corpo **p** é imediatamente executado quando a construção "**trap**" inicia. A sua execução continuará até o término de **p** ou até a execução de uma construção "**exit T**" introduzida no corpo **p**. O término do corpo **p** leva ao término da construção "**trap**". A saída por um "**exit**" leva o "**trap**" a terminar imediatamente, abortando **p** por preempção fraca, do tipo "**weak abort**.

No caso da existência de um tratador de exceção, o controle é imediatamente transferido para o mesmo, após a execução do "**exit**". A ativação do tratamento da exceção é realizado pela construção "**handle**", permitindo o inicio imediato de **q** após o corpo **p** ter sido abortado pelo "**exit**" do "**trap**":

```
trap T in
        p
handle T do
        q
end trap
```

4.3.6 Testes de presença

Além de manusear sinais a partir de construções de preempção, é possível realizar testes de presença de sinais com a construção "**present S else p end present**". Este teste permite decidir entre várias ramificações de programa, em função do valor instantâneo do sinal **S**. Se o sinal **S** está presente, a construção "**present**" termina a seguir, caso contrário a ramificação "**else**" é imediatamente iniciada; a claúsula "**then**" é omitida mas poderia ser usada no lugar de "**else**" se o teste fosse sobre a ausência de sinal ("**not S**").

Para aumentar o poder de expressão, expressões de sinais $\mathbf{e_i}$ e construções condicionais múltiplas ("**case**") podem ser utilizadas em testes de presença:

```
present
    case e1 do p1
    case e2 do p2
    case e3 do p3
    else q
end present
```

4.3.7 Módulo

O módulo é a unidade básica para construir programas Esterel. Ele tem um nome, uma declaração de interface e um corpo que é executável.

```
module <nome> :
        <declaração de interface>
        <corpo>
end module
```

O estilo de programação Esterel permite construir um módulo com nomes padrão para os sinais de entrada e saída. O módulo pode utilizar submódulos que são módulos instanciados pela construção executável "**run**" e cujas entradas e saídas podem ser renomeadas explicitamente usando o símbolo "/". O comportamento do submódulo é o mesmo do módulo, substituindo os nomes padrão (à direita de "/") pelos nomes reais dos sinais (situados à esquerda de "/"). Em nenhum caso é permitido recursividade sobre a instanciação.

Uma forma alternativa de programar o módulo ABCRO a partir do uso do módulo ABRO como módulo genérico é apresentada a seguir como exemplo:

```
module ABCRO:
input A, B, C, R;
output O;
signal AB in
        run ABRO [signal AB / O]
||
        run ABRO [signal AB / A , C / B]
end signal
end module
```

O submódulo ABRO [signal AB/O] se comporta como "loop [await A || await B]; emit AB each R"; o submódulo ABRO [signal AB/A, C/B] como "loop [await AB || await C]; emit O each R"; a difusão instantânea do sinal AB nos dois submódulos paralelos torna o comportamento do módulo ABCRO equivalente ao comportamento já descrito anteriormente "loop [awaitA || await B || await C]; emit O each R". Constata-se ainda que o modelo de programação síncrona fornece como resultado adicional interessante, a reinicialização simultânea dos dois submódulos pelo sinal R.

A declaração de interface do módulo define os objetos que este importa ou exporta: *objetos de dados* (tipos, constantes, funções, procedimentos, tarefas) declarados de forma abstrata em Esterel e implementados externamente e *objetos de interface reativa*: os sinais e sensores. Todos os dados são globais ao programa. Cada dado usado num módulo, deve ser declarado nele.

4.3.8 O conceito de tempo no modelo de programação

No modelo de programação síncrono que suporta a linguagem Esterel, a noção de tempo físico não existe; o tempo físico é visto como um sinal entre outros. Qualquer sinal pode ser considerado para definir uma unidade de tempo independente. O tempo é dito multiforme, i.e. qualquer sinal repetido pode ser considerado como definindo sua própria medida de tempo. O estilo de programação Esterel requer o reconhecimento de todas as unidades de tempo da aplicação e o entendimento de como estas se relacionam entre si. Para representar esta noção, poderia se imaginar uma representação gráfica na forma de vários eixos de tempo com unidades diferentes correspondentes aos diversos sinais repetidos. No módulo Medidor-Velocidade, as unidades de tempo são **Centímetro** e **Segundo** com eixos de tempo próprios; neste caso simples, a relação entre os mesmos permite que se calcule a velocidade.

4.4 Um exemplo ilustrativo do estilo de programação

O estilo de programação é ilustrado a seguir por um pequeno exemplo que descreve o treinamento de um corredor e cuja especificação é:

"*Cada manhã, o corredor faz um número fixo de voltas num estádio. A cada volta, ele corre devagar durante 100 metros, depois ele pula a cada passo durante 15 segundos e termina a volta correndo rápido*".

Num primeiro tempo, a partir da especificação informal anterior, determina-se os sinais que corresponderão as unidades de tempo. Os sinais de entrada são **Manha**, **Volta**, **Metro**, **Passo** e **Segundo**. Os sinais de saída são **Correr-Devagar**, **Pular** e **Correr-Rápido**. Relações entre sinais são estabelecidas: **Manha e Segundo** são sincronizados bem como **Volta** e **Metro.** Os sinais **Correr-Devagar** e **Correr-Rápido**

serão emitidos de forma contínua; o sinal de saída **Pular** será emitido em reação ao sinal de entrada **Passo**. Desta forma fica definida a interface do módulo **Corredor** cujo o código será apresentado a seguir. As relações de sincronização entre sinais são expressas no programa pelo símbolo "=>".

O corpo do programa é uma tradução natural da especificação informal seguindo o estilo de programação apresentado anteriormente cujo o princípio fundamental consiste em controlar as atividades a partir das construções de preempção. O uso de preempções aninhadas é uma forma simples e natural para expressar prioridades entre estas. Uma linha de conduta para uma boa programação consiste a partir de uma primeira leitura da especificação, em construir inicialmente a estrutura de preempções aninhadas, usando a indentação das construções de preempção para visualizar o aninhamento e depois, a partir de uma releitura da especificação, em introduzir as outras construções no corpo do programa. O código em Esterel do módulo Corredor é o seguinte:

```
module Corredor
constant Número-Voltas: integer;
input Manha, Volta, Metro, Passo, Segundo;
relation Manha => Segundo,
  Volta => Metro;
output Correr-Devagar, Pular, Correr-Rápido;
every Manha do
        abort
                abort
                        abort
                                sustain Correr-Devagar
                        when 100 Metro;
                        abort
                                every Passo do
                                        emit Pular
                                end every
                        when 15 Segundo;
                        sustain Correr-Rápido
                when Volta
        when Número-Voltas Volta
end every
end module
```

Para emitir um sinal de forma continua (caso de **Correr-Devagar** e **Correr-Rápido**), utiliza-se a construção "**sustain S**"; a construção fica ativa para sempre e emite **S** a cada instante.

Destaca-se ainda que poderia ser utilizado também a construção "**loop ... each ...**" em algumas situações como por exemplo em "**loop ... each Volta**" no lugar de "**abort ... when Volta**".

Nota-se que se a volta for inferior a 100 metros, o corredor apenas correrá devagar e se esta for mais curta que 100 metros mais 15 segundos, o corredor nunca correra rapidamente; também não é necessário que cada volta seja dimensionada de forma igual.

Ao acrescentamos à especificação anterior do treinamento do corredor o seguinte:

"*Durante a fase na qual o corredor pula, o coração dele é monitorado e em caso de alguma anomalia ser constatada, o corredor parará e retornará ao vestiário*".

O novo código que leva em conta esta especificação adicional toma a seguinte forma:

```
every Manha do
    trap Anomalia in
        abort
            abort
                abort
                    sustain Correr-Devagar
                when 100 Metro;
                abort
                    every Passo do
                        emit Pular
                    end every
                ||
                    <Monitorar-Coração>
                when 15 Segundo;
                sustain Correr-Rapido
            when Volta
        when Número-Voltas Volta
    handle Anomalia do
        <Voltar-Vestiário>
    end trap
end every
```

A atividade **<Monitorar-Coração>** não é descrita aqui mas deverá obrigatoriamente conter uma construção de levantamento de exceção do tipo "**exit Anomalia**" quando esta for constatada; a atividade de tratamento de exceção **<Voltar-Vestiário>** também não é objeto da nossa descrição.

4.5 A assíncronia na linguagem Esterel: a execução de tarefas externas

Procedimentos externos chamados pela construção "**call**" são considerados como instantâneos. Entretanto, é possível controlar ainda a execução de tarefas externas que levam tempo usando o mecanismo "**exec**". Essas tarefas se comportam como procedimentos a serem executados de forma assíncrona com o programa Esterel que terá apenas uma visão lógica destas, se interessando apenas pelo inicio ou fim das mesmas ou ainda, pela suspensão ou preempção desta tarefas através de construções

Esterel. A forma de assincronismo assim introduzida é restrita para permitir a sincronização com a tarefa apenas quando do término da mesma. Estas tarefas não são limitadas ao seu sentido computacional mas podem ser também atividades de um objeto real de natureza física.

A construção "**exec Task (<parâmetros-referência>) (<parâmetros-valores>) return R**" é que permite a execução de uma tarefa externa. Nesta construção, **R** é um sinal de retorno que se restringe a um único "**exec**". Como qualquer sinal de entrada no programa Esterel, cada **R** deve ser declarado na interface de sinal do módulo, utilizando a palavra chave "**return**" no lugar de "**input**". No inicio da construção "**exec**", o ambiente é sinalizado instantaneamente para que a tarefa inicie; o programa Esterel continue a reagir aos sinais de entrada, exceto na "thread" que disparou o "**exec**" que deve esperar o término da tarefa correspondente. Na recepção do retorno **R**, são atualizados instantaneamente os argumentos de referência em todo o programa Esterel, com os valores retornados e a construção "**exec**" termina instantaneamente. Uma operação "**exec**" pode ser suspensa ("**suspend**") ou abortada ("**abort**", "**weak abort**" ou "**trap**") durante sua execução e, em qualquer destas situações, a tarefa externa receberá a sinalização correspondente.

No caso da preempção fraca:

```
weak abort
        exec TASK (X) (...) return R;
when I
```

se **R** ocorre antes ou simultaneamente com **I**, **X** é atualizado e a construção "**weak abort**" termina; se **I** ocorre antes de **R**, a execução de **TASK** e consequentemente da tarefa externa é abortada e **X** não será atualizado.

No caso da preempção forte:

```
abort
        exec TASK (X) (...) return R;
when I
```

se **R** ocorre antes de **I**, **X** é atualizado e a construção "**abort**" termina; se **I** ocorre antes de **R**, a execução de **TASK** e consequentemente da tarefa externa é abortada e **X** não será atualizado; se **R** e **I** ocorre simultaneamente, a construção "**abort**" termina e **X** não é atualizado porque, apesar da tarefa ter sido concluída, o corpo do "**abort**" não recebe o controle.

Se for necessário saber se a tarefa externa terminou num dado instante, o teste "**present R then ... else ... end present**" quando usado na claúsula "**do**" do "**when ...**"

fornece esta informação.

No caso da suspensão:

```
suspend
        exec TASK (X) (...) return R;
when S
```

se **S** ocorrer após o instante inicial, a construção "**exec**" é suspensa e um sinal de suspensão especifico para cada implementação de tarefa é enviado ao ambiente. A terminação de "**exec**" pode ocorrer somente quando a construção é ativa; se **R** e **S** ocorrem simultaneamente, **R** não provocará o fim de "**exec**" e sua ocorrência será perdida. É possível também neste caso, introduzir um teste de presença de um sinal de retorno.

Quando for necessário o controle de várias tarefas simultaneamente, utiliza-se:

```
exec
        case T1 (...) (...) return R1 do p1
        ...
        case Tn (...) (...) return Rn do pn
end exec
```

Quando um "**exec**" múltiplo inicia, todas as tarefas são disparadas simultaneamente; quando pelo menos um sinal de retorno ocorre, a construção "**exec**" termina instantaneamente, abortando as outras tarefas, atualizando apenas o argumento referência da tarefa que terminou; em caso de preempção ou de suspensão todas as tarefas são abortadas simultaneamente.

Numa construção "**exec**" embutida numa malha:

```
loop
        exec TASK (X) (...) return R;
each I
```

se **I** ocorre antes da finalização da tarefa, o programa Esterel sinaliza ao ambiente o aborto da instância corrente da tarefa e nova instância desta é iniciada. Nunca pode haver duas instâncias de uma tarefa disparadas por um mesmo "**exec**" que possam ser ativas no mesmo instante – a não ser em situações onde a construção "**exec**" é abortada e reinicializada, instantaneamente.

4.6 O Ambiente de Ferramentas Esterel

A linguagem de programação Esterel para sistemas reativos vem sendo desenvolvida desde 1982 por equipes do INRIA e da Ecole des Mines de Sophia-Antipolis (France) e se encontra atualmente na sua versão 5 . Seu compilador (atualmente V5.21) esta sendo disponibilizado gratuitamente no site http://www-sop.inria.fr/meije/esterel/ para arquiteturas Solaris, Linux, AIX, OSF1 e Windows NT.

Esta versão do compilador usa uma técnica de compilação baseada em Diagramas de Decisão Binários (BDD) e permite gerar uma implementação de software de um programa reativo na forma de uma máquina de estados finita (FSM) ou uma implementação de hardware na forma de circuitos, obtidos a partir de sistemas de equações booleanos. O código gerado (por exemplo em C) pode ser embutido como um núcleo reativo num programa maior que trata da interface do núcleo e processa os dados da aplicação. Da mesma forma, nas implementações de hardware, a lista de "*gates*" gerada pode ser embutida em circuitos maiores. Além deste compilador, existe um conjunto de ferramentas disponíveis no mesmo site para desenvolver e para verificar programas em Esterel.

Uma das grandes vantagens do modelo usado em Esterel é de favorecer uma abordagem do tipo "o que você prova é o que você execute" (WYPIWYE "*What You Prove Is What You Execute*") [Ber89]. Este aspecto é determinante para que o modelo síncrono se diferencie de grande parte das outras abordagens que, para o desenvolvimento de programas, necessitam de um processo de tradução para passar da especificação validada ao programa implementado. Nesta tradução, são introduzidos detalhes de implementação que, geralmente, são fonte de erros ou de modificações no comportamento já verificado.

No que se refere a validação em linguagens síncronas, as principais propriedades a serem verificadas para os sistemas de tempo real são: "vivacidade" ("*liveness*") que pode ser entendida como "algo útil deverá acontecer em algum momento"; e "segurança" ("*safety*") que corresponde a "algo ruim nunca deverá acontecer" ou "algo útil deverá acontecer antes de algum tempo" no caso dos sistemas de tempo real.

A verificação consiste em comparar um programa com as especificações desejadas de algum aspecto ou da totalidade das mesmas. A verificação das propriedades citadas se faz sobre a estrutura de controle do programa, os dados não influem diretamente. Duas abordagens de verificação podem ser diferenciadas, conforme as especificações se apresentem no mesmo formalismo que o programa (por exemplo, autômato) ou em formalismos diferentes (por exemplo, autômato para o programa e lógica temporal para as especificações). No caso da primeira abordagem, pode se utilizar duas técnicas diferentes: a de verificação por equivalência e a de verificação por observadores.

A técnica de verificação por equivalência usa a noção de bisimulação fraca que foi definida por Park e usado por Milner [Arn94]. A bisimulação é uma relação binaria entre estados de dois sistemas de transições (no caso similares aos autômatos) que

define a equivalência entre estes. A verificação da equivalência consiste em verificar a existência desta relação de bisimulação. O primeiro passo consiste em distinguir os sinais relevantes para o aspecto da especificação que se pretende verificar, dos irrelevantes que passam a ser considerados como eventos internos. O segundo passo consiste em reduzir na sua forma mínima o autômato na forma obtida anteriormente, do ponto de vista da bisimulação fraca. Este autômato reduzido contém poucos estados e transições, podendo ser facilmente verificado as propriedades por simples leitura ou a sua equivalência com um autômato das especificações.

A técnica de verificação por observadores consiste em construir um observador **O** que é um outro programa reativo que expressa a propriedade a ser verificada e a colocá-lo em paralelismo síncrono com o programa **P** a verificar e com outro programa reativo **E** que representa o ambiente e gera seqüências de entrada para os dois anteriores. Como o papel do observador **O** é de ver o comportamento do programa **P**, ele tem ainda como entrada, as saídas do programa **P**. Técnicas de analise de alcançabilidade para autômatos permitem verificar o programa, detectando eventuais diferenças entre o comportamento do programa e aquele descrito pelo observador. Qualquer linguagem síncrona pode ser usada para programar o observador **O** e o ambiente **E**.

A verificação pela abordagem da lógica temporal consiste em verificar se formulas de lógica temporal que representam as propriedades desejadas são satisfeitas ou não pelo programa. Uma das formas de construir um verificador neste caso consiste em construir também um observador com as propriedades expressas em lógica temporal e compilado como programa Esterel. Existem na literatura, outras formas de expressar as formulas de lógica temporal que resultaram em varias ferramentas de verificação baseadas nesta técnica.

As ferramentas incluídas no ambiente de desenvolvimento Esterel permitem os três tipos de verificação descritas anteriormente.

O ambiente de desenvolvimento e verificação Esterel, **Xeve** disponibilizado em http://www-sop.inria.fr/meije/verification/Xeve/ é um ambiente com interface gráfica que permite a análise e a verificação de programas Esterel baseado na representação em máquinas de estados finitos. As ferramentas de **Xeve** são:

- **Hurricane** (disponível para Solaris e Linux) que é um verificador de modelos (*model checking*) sobre os programas Esterel, baseados em fórmulas de lógica temporal. As fórmulas de lógica temporal são transformadas em observadores Esterel usando um tradutor tl2strl. Esta ferramenta gera um novo programa Esterel obtido a partir do programa principal e das fórmulas e onde, as entradas são as do programa principal e as saídas são as do programa principal e as obtidas das fórmulas. A ferramenta verifica o estado dos sinais de saída e o resultado obtido indica para cada saída se a mesma pode ser emitida ou não.
- **fc2symbmin** que analisa as máquinas de estado finitos geradas pelo compilador Esterel e descritas no formato FC2 – formato definido pela

equipe de pesquisa deste projeto para facilitar a interface com outros software de verificação –. Esta análise inclui:

- a minimização de FSMs (máquinas de estados finitos) obtida a partir da noção de bisimulação dos autômatos reativos obtidos do programa Esterel,
- o diagnóstico através de observadores que são programas inicializados em paralelo com o programa principal, testando seus sinais, interagindo com ele e gerando sinais de falha ou sucesso; esses observadores podem ainda gerar sinais para simular o ambiente Esterel,
- o produto síncrono de FSMs para implementar o operador paralelo síncrona de Esterel,
- a ocultação de sinais para reduzir o tamanho da FSM e a complexidade da análise, e
- a verificação de equivalência que determina se dois autômatos podem se comportar da mesma forma ou não tendo o mesmo ambiente.

- **Atg** disponível em http://www-sop.inria.fr/meije/verification/index.html é um editor gráfico de autômato que permite a visualização da máquina de estado finitos minimizada.
- **Xes** que é um simulador gráfico habitualmente distribuído com o compilador para lhe servir de depurador (“debugger”) e que permite executar seqüências de execução como por exemplo as extraídas do diagnostico de verificação de modelo (*model checking*).

As figuras 4.2 a 4.4 apresentam algumas das telas do ambiente Xeve.

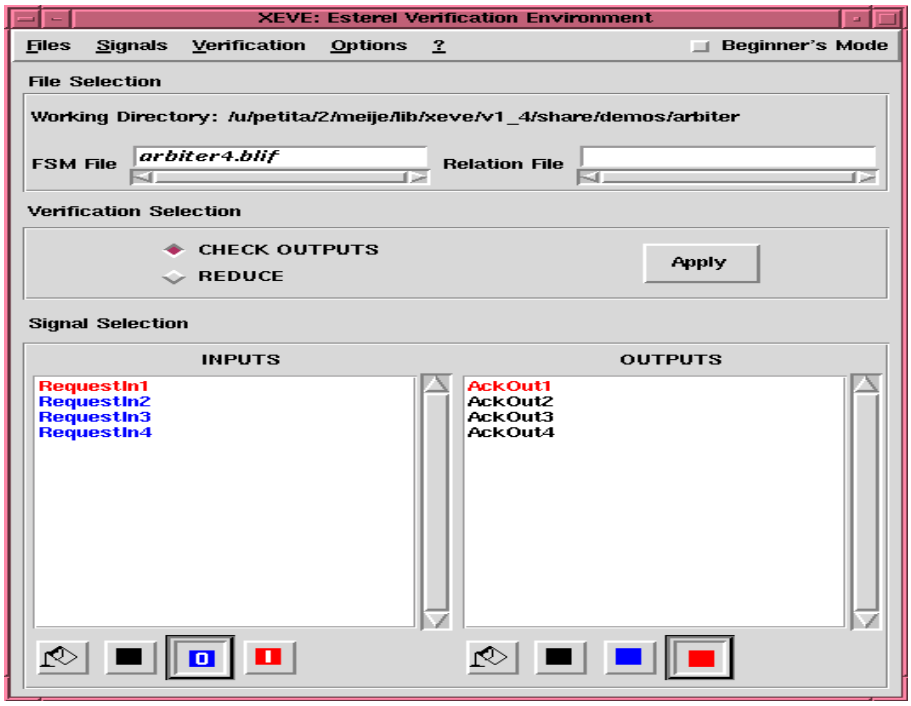

Figura 4.2 - Tela Principal do Ambiente XEVE

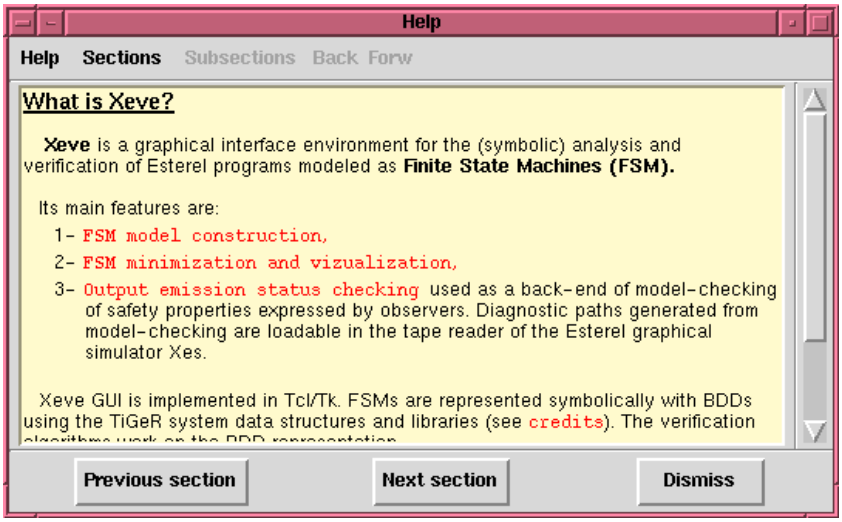

Figura 4.3 - Tela de Ajuda do Ambiente XEVE

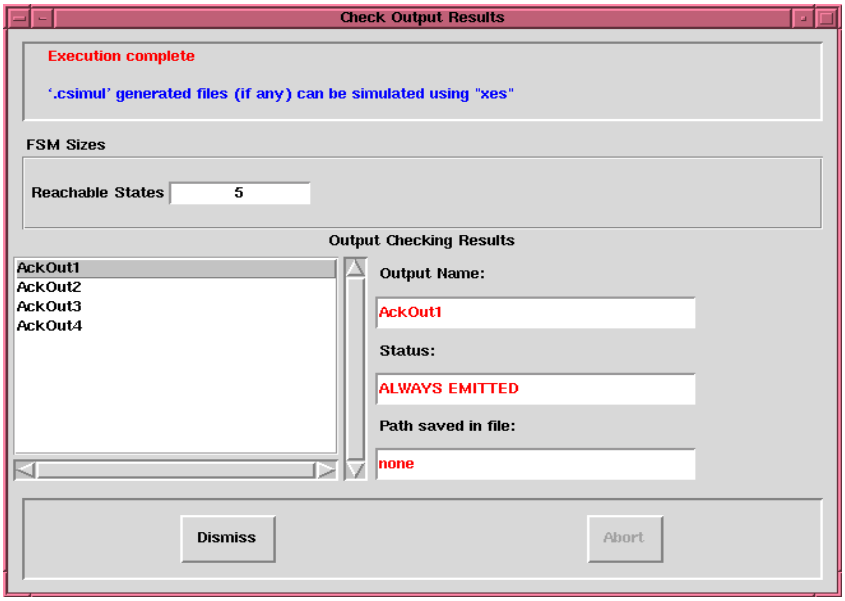

Figura 4.4 - Tela de Resultados do Ambiente XEVE

4.7 Implementações de programas em Esterel

Um programa em Esterel pode ser implementado de duas formas diferentes: por software usando um autômato ou por hardware usando circuitos booleanos seqüenciais. Neste livro, apresentaremos somente a primeira forma de implementação; o leitor que tiver interesse na segunda forma de implementação, deverá recorrer as seguintes referencias básicas [Ber92], [STB96].

- **Implementação usando um autômato**

Uma das características mais interessante do modelo síncrono e da linguagem Esterel é a capacidade de produzir um autômato (ou máquina de estados finitos), construído pelo compilador a partir das regras da sua semântica de execução que à cada evento de entrada, associa uma transição vindo de um estado.

O autômato resultante da compilação é determinista: o paralelismo e as comunicações de sinal em código Esterel desaparecem e são seqüencializados dentro de cada transição interna, produzindo um código seqüencial. Este tipo de compilação em autômatos oferece vantagens do ponto de vista da eficiência e em relação à previsibilidade, tão necessária em sistemas de tempo real.

A principal vantagem reside na eficiência na execução. Os sinais locais usados para comunicação interna entre construções desaparecem no autômato, como os nós não-terminais intermediários desaparecem durante a geração de “parser” numa compilação; os sinais locais podem em conseqüência ser considerados como tendo verdadeiro atraso zero em tempo de execução. Além do mais, como não há mais nenhum paralelismo, não há custo adicional em tempo resultando do gerenciamento em tempo de execução de tarefas ou da comunicação e sincronização de tarefas em tempo de execução, em contraposição com o que ocorre com o uso de outras linguagens concorrentes que não suportam o modelo síncrono e que necessitam de usar um suporte para gerenciamento e comunicação de tarefas, com o custo adicional e a não-previsibilidade que este proporciona.

Como vantagem adicional desta compilação em autômato, destaca-se a previsibilidade do tempo de transição máxima de um autômato, de especial relevância quando se trata de Sistemas de Tempo Real.

Entretanto a desvantagem de um autômato esta associada ao seu tamanho. Aplicações que resultam em autômatos relativamente pequenos como em protocolos, interface homem-máquina, controle de processos simples são facilmente manuseáveis, o que não é o caso em algumas aplicações mais complexas na qual a explosão de estados pode se tornar relevante.

No ambiente de desenvolvimento para a linguagem Esterel, o autômato é produzido num formato de saída intermediário, chamado OC que é comum à linguagem Lustre já citada e que pode ser traduzido numa linguagem alvo de uso geral tal como C, Ada, Java.

4.8 Discussão sobre o modelo e a linguagem

A abordagem adotada para tratar a reatividade nos modelos síncronos e em particular em Esterel se baseia na hipótese de sincronismo que determina a instantaneidade da reação. A cada reação, o sinal vindo do ambiente ou de um componente se encontra presente em todos os componentes paralelos. Esta abordagem pode levar a situações indesejáveis nas quais um sinal pode ser alterado devido ao teste sobre o mesmo sinal, resultando num problema de causalidade e consequentemente em programas incorretos. A seguir são apresentados alguns exemplos de situações que geram programas incorretos e que necessitam da definição de uma semântica formal de execução para resolve-los.

- **Causalidade:**

A expressividade da linguagem síncrona Esterel possibilita a construção de programas não-causais.

A capacidade de comunicação através de sinais difundidos instantaneamente, pode levar alguns programas a desrespeitar a causalidade. Por exemplo, o programa que segue:

```
present S else emit S end
```

corresponde a testar a ausência de um sinal **S**; se o mesmo se fizer ausente, o sinal é emitido e então presente; se estiver presente, o sinal não é emitido e consequentemente ausente. Este programa apresenta uma incoerência com o principio de comunicação com difusão e deve ser absolutamente rejeitado em tempo de compilação. O circuito lógico equivalente a este programa "**S = not S**" mostra facilmente o absurdo deste.

```
present S then emit S end
```

Da mesma forma, o programa é não determinista, pois **S** é presente se e somente se **S** for emitido, o que não determina seu estado. O circuito lógico equivalente a este programa "**S = S**" também não faz sentido.

O programa seguinte:

```
present S1 else emit S2 end
||
present S2 else emit S1 end
```

expressa as seguintes situações: $\mathbf{S_1}$ ausente e $\mathbf{S_2}$ presente; $\mathbf{S_2}$ ausente e $\mathbf{S_1}$ presente. Este programa fere o determinismo do modelo síncrono porque essas duas situações não podem ocorrer simultaneamente e, portanto, também deve ser rejeitado.

Podem ainda existir situações nas quais apesar de ser construído a partir de componentes paralelos sem problemas de causalidade, o sistema total pode apresentar incoerências do ponto de vista da causalidade. Em [BoS96] é apresentado o seguinte exemplo desta situação. Sejam dois programas p_1 e p_2 respectivamente que apresentam o mesmo comportamento (U emitido e T é emitido se S é presente):

```
present S then emit T end; emit U
```

e

```
emit U; present S then emit T end
```

o sistema total que consiste em colocar cada um destes programas em paralelo com um terceiro programa p_3 cujo código é:

```
present U then emit S end
```

apresenta uma solução correta para o caso $p_2 \| p_3$ e uma não-causalidade para $p_1 \| p_3$.

- **Malhas instantâneas:**

Uma outra situação indesejável que pode levar a uma situação sem solução é o caso das malhas instantâneas cujo corpo termina no instante onde é executado pela primeira vez; por exemplo na instrução:

```
loop x := x+1 end
```

- **Sinais com valores:**

Algumas situações envolvendo sinais com valores geram incoerências no programa. Por exemplo a instrução

```
emit S(?S + 1)
```

indica a emissão de um sinal **S** cujo o valor é fornecido por **?S + 1** que corresponde ao valor corrente de **S** acrescido de **1**; o efeito obtido é equivalente a uma realimentação positiva e é inaceitável num programa Esterel.

Todas essas situações paradoxais devem ser evitadas pelo programador ou detectadas e rejeitadas a partir de uma analise estática dos programas pelos compiladores da linguagem Esterel . Uma forma fácil consiste em proibir a auto-dependência estática dos sinais, da mesma forma que se exige que circuitos lógicos sejam acíclicos; mas esta forma se apresenta como restritiva para algumas situações raras nas quais pode se desejar programas com dependências cíclicas como no caso do mecanismo de arbitro de barramento simétrico numa rede local, conforme descrito em [Ber 99].

4.9 Conclusão

Este capítulo foi escrita a partir da bibliografia seguinte: [Ber89], [BeB91], [GLM94], [BoS96] na apresentação e discussão da abordagem síncrona e [BoS91], [BeG92], [Ber98], [Ber99] na apresentação da linguagem síncrona Esterel.

A abordagem síncrona apresentada neste capítulo adota o modelo síncrono que, basicamente, assume reações instantâneas a eventos externos. Esta hipótese de reações instantâneas pode ser entendida como qualquer tempo de resposta menor que o tempo mínimo entre eventos vindos do ambiente. Muito aplicações de tempo real sustentam esta hipótese como verdadeira.

A utilização de uma linguagem síncrona, neste livro Esterel, leva à implementações

eficientes dos sistemas de tempo real baseadas em autômatos ou em circuitos booleanos. Essas implementações são facilmente validadas segundo o principio “o que você prova é o que você execute” descrito anteriormente.

Sistemas embutidos, protocolos de comunicação, “*drivers*” para periféricos, sistemas de supervisão e controle, e interfaces homem-máquina são aplicações ou partes de aplicações mais complexas para as quais o uso da abordagem síncrona é particularmente adaptada. No capitulo 5, a partir de um exemplo simples será apresentada uma metodologia de programação baseada na abordagem síncrona e seguindo o estilo de programação da linguagem Esterel.