UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Ferramentas para a linguagem de especificação LOTOS

autor: Humberto Maia Lima.

orientador: Maurício Ferreira Magalhães

Tese apresentada à Faculdade de Engenharia Elétrica
da Universidade Estadual de Campinas,
como parte dos requisitos exigidos
para obtenção do título de mestre em
Engenharia Elétrica

Agradeço a todos que ,
direta ou indiretamentamente,
influíram neste trabalho e
em especial a Cristina Zabeu
pela sua valiosa colaboração.

Sumário

Este trabalho propõe a implementaçao de um ambiente de auxílio ao desenvolvimento de especificações de sistema que utilizam LOTOS como linguagem. Tal ambiente é formado de um conjunto integrado de ferramentas composto de um editor de texto, um analisador sintático e semântico e um simulador para LOTOS. Diferente de outras implementações descritas na literatura, é proposta a implementação do simulador LOTOS em uma linguagem orientada para objetos utilizando um mapeamento da semântica dos operadores LOTOS para classes C++.

Abstract

An environment for system design using LOTOS as specification language is proposed. The environment is composed by an editor, a syntax and semantic checker and a simulator all working in an integrated form. Different from similar works described in the literature, the proposed implementation of the LOTOS simulator is based on an object oriented language mapping the semantic of LOTOS operators to C++ classes.

Conteúdo

Capítulo 1

Introdução

O processo de desenvolvimento de software vem sofrendo evoluções contínuas, à medida que os sistemas que necessitam de soluções computacionais se tornam mais complexos. Nestas evoluções, o conceito de programação variou de uma atividade artística, pessoal, para um trabalho com regras definidas [Men 88]. Utilizam-se para estas regras tanto conceitos matemáticos formais (uma vez que um programa tenta resolver um problema, o que é uma atividade matemática), quanto visões de engenharia, onde é importante obter resultados visíveis rapidamente, utilizando muitas vezes abordagens empíricas.

O desenvolvimento de um sistema se inicia com uma análise do que se deseja obter. Este processo pode ser mais ou menos complexo, levando em consideração o reaproveitamento de experiências passadas. É virtualmente impossível conseguir que uma solução de software satisfaça totalmente todas as necessidades de um sistema. Em geral, o problema vai sendo "refinado" em etapas e em cada uma são utilizados conhecimentos diferentes. Por exemplo, ao especificarmos os requisitos do sistema, estaremos relatando apenas o comportamento externo do sistema, os seus "contornos". A cada etapa são utilizadas também diferentes "linguagens" para expressar os problemas. As primeiras definições são feitas, em geral, em linguagem natural, enquanto, durante a evolução das etapas, aparece a necessidade de se utilizar linguagens mais formais.

É na etapa onde a especificação do sistema a ser desenvolvido deve ser mais precisa que são utilizadas técnicas formais de descrição do software. A "formalização" é feita utilizando métodos matemáticos; desta maneira, o vocabulário, a sintaxe e a semântica de uma especificação ficam definidas de forma única. Dentro do processo evolutivo de ''construção" de software, as técnicas formais são complementares às utilizadas nas fases iniciais do desenvolvimento. Algumas das vantagens obtidas na utilização de técnicas formais são [Som 89]:

- a possibilidade de verificação de programas com relação a uma especificação preliminar
- o processamento automático da especificação, auxiliando o desenvolvimento do produto
- a simulação da especificação

Entretanto, especificações formais não são ainda uma prática corrente no desenvolvimento de software, por fatores como:

- o caráter "conservador" das empresas, onde se prefere não alterar o processo de desenvolvimento sem noção de quanto isto representaria em lucros ;
- falta de familiaridade com métodos matemáticos, o que gera a impressão de "ser difícil de usar";

- não existência de métodos formais para classes de problemas difíceis de modelar;
- falta de ferramentas de apoio às técnicas escolhidas.

As técnicas de especificação formal surgiram em função de estudos de técnicas formais de verificação de programas. Os métodos formais pretendem aumentar a qualidade do software desenvolvido de duas formas :

- fornecendo uma especificação clara, completa, não ambígüa e fácil de validar matematicamente [1]
- Possibilitando verificações efetivas durante o processo de produção de software

1.1 Técnicas de descrição formal

Utiliza-se o nome "técnica de descrição formal" para identificar as técnicas de desenvolvimento de software que usam métodos formais. Atualmente, o uso destas técnicas tem se concentrado na fase de especificação de sistemas. Algumas das características desejáveis em uma técnica de descrição formal (ou TDF) são:

1 onde validar significa mostrar que o que está sendo feito é o que se desejava fazer

- poder de expressão
- facilidade para abstrações
- modularidade
- base matemática

Em um desenvolvimento "formal" constrói-se uma especificação que fornece uma descrição matemática do que o programa deve fazer. Esta descrição vai sendo "trabalhada" segundo alguns passos até chegar a uma forma executável. Entretanto, mesmo com o projeto na fase de especificação, podemos validá-lo derivando propriedades, baseadas no modelo matemático que o descreve e mesmo "animá-lo" .

Existem várias técnicas propostas para auxílio ao desenvolvimento de especificações de sistemas, cada uma apresentando características próprias. Técnicas que descrevem propriedades seqüenciais e algorítmicas de programas como HOL (*High Order Logic*) e VDM (*Vienna Development Method*) [Ped 88] baseiam-se em cálculo de predicados. Métodos baseados em álgebra de processos como CSP (*Communicating Sequencial Process*) e CCS (*Calculus of Communicating Systems*) procuram expressar concorrência. Métodos gráficos, como SDL e redes de Petri, possibilitam o modelamento de máquinas de estado finitas estendidas [Bloom 88].

LOTOS é uma linguagem de especificação baseada em álgebra de processos desenvolvida pela ISO para especificar serviços e protocolos ISO [ISO 87] [Eij 89]. LOTOS é derivada de duas linguagens já existentes: CCS, de onde LOTOS herdou suas expressões de comportamento para representar a parte de controle, e ACT ONE para descrição de tipos de dados abstratos.

1.2 O propósito da tese

Este trabalho propõe-se a descrever, no âmbito de ferramentas para desenvolvimento de sistemas, a implementação de um conjunto de facilidades para a especificação de sistemas utilizando LOTOS como linguagem. Acreditamos que o uso de linguagens formais na fase de especificação de um sistema proporciona um nível de abstração adequado para a definição da solução que o sistema propõe resolver. Nesta fase há pouco comprometimento com o ciclo de produção de código executável e, portanto, podemos ainda corrigir ou alterar conceitos sem "efeitos colaterais" graves. A possibilidade de validação antecipada, através de simulações ou protótipos, corrige distorções entre as visões de quem define e de quem implementa o sistema. E a fase "árdua" de depuração de programas pode ser antecipada pela verificação matemática da especificação, reduzindo o tempo gasto para corrigir falhas com o produto "pronto".

Hoje em dia, os métodos formais são utilizados na descrição de propriedades seqüenciais de sistemas de pequeno porte, e na definição de protocolos. Questões como propriedades temporais e paralelismo puro ainda são objeto de estudos, e resultados estão surgindo a cada dia. O desenvolvimento de ferramentas automatizadas para suportar o processo de desenvolvimento utilizando linguagens formais é um ponto fundamental para sua difusão nos meios de produção de software.

O conjunto de facilidades descrito neste trabalho é composto de um editor de textos, um analisador sintático e semântico para LOTOS e um simulador que permite uma verificação dinâmica do comportamento de uma especificação em LOTOS. Essas facilidades foram implementadas na forma de ferramentas software que podem ser utilizadas dentro de um ambiente integrado onde o usuário efetua o ciclo *edição-compilação-simulação* dentro do mesmo ambiente sem necessidade explícita de geração de passos intermediários. Da mesma forma, pode-se utilizar as ferramentas isoladamente através de arquivos de dados intermediários para passagem de informação entre as diversas ferramentas.

Este trabalho está organizado da seguinte maneira : no próximo capítulo é analisado o uso de ferramentas nas diversas etapas que compoem o processo de desenvolvimento. Em seguida, no capítulo 3, é feita uma introdução a LOTOS como linguagem de especificação. No capítulo 4 é apresentado um

modelo de um conjunto de ferramentas integradas que podem ser usadas como auxílio na construção de especificações de sistemas utilizando a linguagem. No capítulo 5 é descrito um programa que implementa a funcionalidade da ferramenta proposta.

produto, contratação de publicidade, até o fornecimento do produto. As etapas de desenvolvimento seriam estruturadas em seqüência, o que significa que uma etapa mal conduzida poderia inviabilizar todo o produto. Em cada etapa, diferentes especialistas teriam participação, e um controle do processo como um todo seria feito por uma gerência, evidenciando que seriam utilizadas formas distintas de expressão do problema ou de suas partes.

Em computação, analisaremos como se aplica esta noção de projeto descrita acima.

2.1.1 Etapas do desenvolvimento

Um modelo que tenha solução computacional já sofreu, antes desta classificação, um enquadramento em certas regras e condições que o tornam "computável". Chamaremos esta fase preliminar de análise de requisitos. Nesta fase, em geral, opinam os "clientes" do problema (os contratantes da solução), os futuros usuários (que nem sempre são os "clientes") e os projetistas de software. Distingue-se, nesta fase, a noção de objetivos e de requisitos do sistema: um requisito é algo que pode ser testado no sistema, enquanto um objetivo é uma característica que o sistema deve oferecer. Um exemplo disto é [Som 89] ter como objetivo um sistema "user-friendly", o que é bastante subjetivo uma vez que classificar um sistema como "amigável"

Capítulo 2

Ferramentas no processo de desenvolvimento

2.1 O processo de desenvolvimento de um projeto

O desenvolvimento de um projeto tem como objetivo solucionar um certo problema. Isto implica em, primeiramente, entender a natureza do problema que se quer resolver. Ao entender a natureza do problema conseguimos ter a noção de um modelo que possa descrevê-lo, e com este modelo podemos identificar diversas etapas do processo de solução do problema. Como um determinado problema não é visto de maneira uniforme pelas várias pessoas por ele afetadas, podemos depreender que as etapas de solução serão relacionadas a estes "clientes", de acordo com seu grau de aproximação com o problema. Eventualmente, serão etapas estruturadas hierarquicamente, ou em um esquema "top-down", ou ainda objetos isolados que "conversam" entre si.

Como exemplo, tomemos um problema que envolve a criação de um produto comercial. A natureza deste problema é fornecer uma mercadoria com determinadas características, com lucro para o fabricante e benefícios para uma dada faixa de pessoas. Um modelo que descreva este problema pode ser estruturado em uma etapa de pesquisa de opinião, uma análise de fornecedores de matéria-prima e da linha de produção da empresa, estudos para melhor produção com menor custo, esquemas de distribuição e vendas do

é bastante particular a um grupo de usuários. Por sua vez, um requisito associado é o de fornecer as opções de comandos aos usuários através de menus. Nesta etapa de desenvolvimento a forma de expressão é, em geral, uma linguagem natural, que possa ser entendida por diversos grupos ligados ao problema. Um fator importante é que o nível de abstração nesta etapa é bastante alto. Observaremos que a noção sobre a resolução do problema vai se tornando mais objetiva no decorrer das etapas do desenvolvimento.

Uma etapa subseqüente a esta é a da especificação de requisitos. Nesta etapa temos um texto mais formal, que define, em maior detalhe, os serviços que o sistema proverá . Um documento que expresse esta etapa deve servir como "acordo" entre o solicitante do sistema e o pessoal técnico que irá desenvolvê-lo. Através de uma especificação de requisitos podemos vislumbrar os contornos do produto que será gerado para resolver o problema, fruto do modelo que se imaginou para explicá-lo.

A próxima etapa é a de se definir como será elaborado este produto. Um produto realizará as funções dele esperadas através de sua estrutura e de seus componentes. Para se definir a forma - estrutura e componentes - de um produto software utilizamos uma especificação funcional, que é ainda uma descrição abstrata, que deve servir como base para o projeto do produto software e para sua implementação. Este documento é produzido em uma

fase onde seus leitores serão os projetistas de software, e portanto pode lançar mão, para sua escrita, de uma linguagem mais formal e precisa.

Nesta última etapa observamos que o produto possui um contorno mais definido; entretanto, o "projetar" é bastante particular do projetista que o fará, e depende de sua prática e de sua experiência prévia. Até esta etapa, o problema, seu modelamento e sua especificação funcional eram independentes de métodos de projeto ou linguagens. Nesta fase, uma vez que o procedimento se torna mais "pessoal", aumenta a importância de se utilizar formas de expressão mais "automáticas" e uniformes.

2.1.2 O uso de ferramentas

Observamos no item anterior, que as várias etapas do processo de desenvolvimento correspondem a níveis diferentes de abstração na solução de um problema. Além disso, o "público" que faz parte de cada etapa é diferente, e a forma como cada etapa é documentada e transmitida também é distinta. Vem daí a necessidade de se organizar o trabalho em cada etapa, aumentando assim produtividade e correção do produto gerado. Para tal, definiremos o conceito de ferramenta. A definição de uma ferramenta estará ligada à etapa de desenvolvimento em que será utilizada; uma ferramenta também pode ser vista como "um conjunto de funções com uma interface com usuário apropriada" [Eij 89].

O uso de uma ferramenta tenta uniformizar a forma como evolui o processo de projeto em uma dada etapa. A ferramenta agrupa um conjunto de funções pertinentes àquela etapa. Este agrupamento pode ser guiado pela dependência funcional entre estas funções, ou por alguma correlação semântica bastante forte, ou apenas porque seria mais consistente com a interface a ser fornecida ao usuário ter aquelas funções operando em conjunto.

Em suma, o uso de uma ferramento deve facilitar a execução de cada fase de um projeto, auxiliando o "público" associado a cada etapa ao qual está relacionada.

2.2 Ferramentas no desenvolvimento de projetos

2.2.1 Definição e características de ferramentas

Como vimos, uma ferramenta deve incrementar a produtividade de uma fase de projeto ao qual está associada, melhorando a qualidade do produto gerado naquela fase. Neste sentido, uma ferramenta deve possuir certas propriedades. Do ponto de vista do usuário, temos que:

- uma ferramenta deve ser "usável", ou seja, deve auxiliar uma parte do desenvolvimento do projeto;

- uma ferramenta deve, na medida do possível, ser integrada, no sentido em que se várias funções devem ser executadas, elas devem estar disponíveis de maneira uniforme para o usuário;

- uma ferramenta deve ser consistente na interface que oferece ao usuário, ou seja, não deve apresentar muitas formas diferentes de efetuar uma mesma operação.

Do ponto de vista do implementador de ferramentas, é necessário delinear quais são as funções que serão providas pela ferramenta. Do ponto de vista computacional uma ferramenta é definida através de:

- uma sintaxe
- uma semântica estática
- semântica dinâmica
- um modelo matemático que a descreva

O implementador de uma ferramenta trabalha sobre estas partes de forma a extrair uma representação que, aplicada sobre o problema que se deseja resolver, possa fornecer ao projetista uma visão uniforme e automatizada do seu problema .

2.2.2 Ferramentas utilizadas nas fases de um projeto

Vimos que as etapas da elaboração de um projeto são associadas a diferentes níveis de abstração do mesmo problema, e que o uso de ferramentas tenta uniformizar e melhorar a qualidade do trabalho. Mas observamos também que as diferentes etapas correspondem a diferentes "espectadores" e, com isso, as ferramentas serão particulares a cada uma destas etapas. Do item 2.1.1 podemos evidenciar três grandes fases de um projeto:

- a fase de especificação de requisitos
- a fase de especificação funcional
- a fase de implementação do produto

Procuraremos, neste item, enumerar algumas das ferramentas que poderiam auxiliar cada uma destas etapas.

2.2.3 Ferramentas na fase de especificação de requisitos

Nesta fase o projeto ainda é descrito em uma linguagem pouco formal, para ser entendida (também) por um público não especializado em computação. Ferramentas nesta fase deveriam fornecer a este público o que uma análise dinâmica fornece; por exemplo, quais seriam os resultados possíveis se um conjunto particular de decisões fosse tomado do ponto de vista técnico. A partir das características de ferramentas descritas no item 2.2.1, seriam ferramentas úteis nesta fase de desenvolvimento:

- um editor que guiasse o especificador em um "roteiro" com uma estrutura de especificação de requisitos, que seria composta de uma introdução, um modelamento do sistema, uma lista dos requisitos funcionais fornecidos ao usuário, uma descrição do hardware (com, pelo menos, a configuração mínima necessária), os requisitos de base de dados, o ambiente em que o produto deve operar, informações sobre manutenção, glossário e índice.

- um editor gráfico que gerasse uma "macro-visão" do sistema, identificando módulos que resolverão partes do problema, e sua inter-relação

- formatadores e "drivers" para impressoras, para que o documento final tivesse boa forma gráfica

- controlador de versões de documentos

2.2.4 Ferramentas para a fase de especificação funcional

Nesta fase ainda teremos uma especificação que vai partir de um texto, mas sobre o qual serão feitos controles mais precisos e formais. Uma especificação, nesta fase, deve ser não-ambígua, abstrata, consistente, clara e concisa. As ferramentas, para suporte a esta fase, utilizam em sua construção conceitos da funcionalidade da linguagem que será utilizada para a especificação; seriam, por exemplo:

- um editor orientado a sintaxe, para auxiliar a escrita do texto da linguagem, fornecendo objetos da linguagem, controlando (possivelmente) operações sobre eles

- um analisador semântico e sintático, para analisar se um dado texto estaria correto sintaticamente e se suas propriedades semânticas se verificariam. Ou seja, uma ferramenta que tanto auxiliaria na análise do produto como um objeto estático (sintaxe) quanto na análise dos resultados possíveis se o texto fosse realmente executado (semântica) [Free 87].

- um verificador, para checar a especificação com relação a um conjunto de requisitos (formais).

- um simulador, para "animar" a especificação e que, eventualmente, agisse como uma ferramenta de prototipação (isto é, observação prévia do comportamento do produto que se está desenvolvendo)

2.2.5 Ferramentas para a fase de implementação

A característica principal da fase de implementação [Eij 89] é mapear o que foi descrito na fase anterior (de especificação) para um outro nível, mais detalhado. Nesta fase sobressai uma grande parcela de "criação" dos projetistas. As operações (ou funções) executadas nesta fase são no sentido de mapear os conceitos descritos nas fases anteriores em "objetos" menores

(mais concisos), decidir sobre o que deve ser implementado e como, e ainda, verificar se estas decisões foram corretas.

Ferramentas para esta fase seriam:

- depuradores
- geradores de testes
- compiladores para um código executável

2.3 Integração de ferramentas

2.3.1 Aspectos de desenvolvimento de projeto

Para o usuário de um ambiente, um conjunto integrado (isto é, que seja auto-contido em todas as suas partes) melhora sua produtividade em relação a um ambiente com ferramentas isoladas, ou sem ferramentas. A tendência nos grandes projetos atuais é de se utilizar ferramentas inteligentes que agrupam outras ferramentas e coordenam todo o ciclo de vida do projeto.

O uso de ferramentas automatizadas não reduz, por si só, os problemas pertinentes ao desenvolvimento de software. É necessária uma quantidade razoável de controle e comportamento sistemático para desenvolver sistemas de algum porte [Free 87] [Hen 80] . Ferramentas, neste contexto, "obrigam" a se respeitar um mínimo de regras. Ferramentas integradas devem apoiar

ainda mais este ponto de vista, devendo ter como "linha mestra" a obtenção de melhores resultados do que com ferramentas separadas.

Ressaltamos que existem diversas perspectivas sob as quais podemos classificar conjuntos de ferramentas; uma possibilidade é a do trabalho conjunto de uma série delas, trabalho este sincronizado com os procedimentos operacionais e os projetistas envolvidos, visando fornecer o máximo de suporte. Outras classificações dividem ferramentas entre aquelas que, mesmo em conjunto, agem separadamente (em um conceito de "toolbox"), e as que, ao contrário, fazem parte de um mesmo ambiente "fixo", onde todas as interações entre as ferramentas são pré-definidas e em um formato estabelecido.

Ferramentas são necessárias em todas as fases de um projeto, e não somente nas fases de implementação. Na elaboração de qualquer sistema, deve-se investir em fazer com que todas as partes do projeto caminhem juntas, e a utilização de ferramentas integradas colabora bastante com esta visão.

2.3.2 Características de ambientes integrados

Fornecer um ambiente realmente integrado não significa apenas construir um conjunto de ferramentas que resolva partes de um problema [Men 88], [Hal 90]. Um ambiente onde não se tenha suporte para o ciclo de vida do software não pode ser chamado como tal. Existem várias ferramentas que fazem este

trabalho (de coordenação e apoio do ciclo de software), mas poucas "conversam" entre si. Uma boa ferramenta (ou um conjunto de ferramentas) deve ser baseada em um bom método - ferramentas serão tão boas quanto forem os métodos que fornecem sua definição. Ou seja, os conceitos básicos do método devem ser modelados, deve haver uma boa representação destas entidades para o usuário, as regras que definem o método devem ser obedecidas pela ferramenta, bem como os procedimento (passos) que o método segue. Uma vez que um único método não consegue englobar todo o ciclo de desenvolvimento de software, as representações dos vários métodos usados neste ciclo devem "conviver", adaptando-se a mudanças (e sendo, portanto, flexíveis).

Integração significa consistência de interfaces, compartilhar uma mesma base de dados (ou fazer com que representações possam servir como fonte uma para as outras), fornecer à equipe de desenvolvimento meios de cooperação, além de possibilitar um controle sobre todo o processo de desenvolvimento. O não uso de ferramentas integradas provoca alguns sintomas bem conhecidos em desenvolvimentos de software como, por exemplo, o desperdício de esforço manual, inconsistências entre as especificações e implementações, a gerência "desinformada" do que acontece na prática, e a proliferação de versões do produto ou de suas partes.

Ferramentas integradas em um único ambiente constituem o que pode ser chamado de ambiente "fechado". Como cada ferramenta é parte do ambiente, a integração entre elas é bastante forte. Usam, por exemplo, a mesma base de dados e a mesma interface para o usuário. Entretanto, o número de ferramentas em um ambiente como este é limitado (uma vez que devem ser feitas pelo fornecedor do ambiente!). Um ambiente com esta característica engloba, em geral, o suporte para apenas poucas partes de um projeto.

Ambientes abertos têm como idéia central o agrupamento de diversas ferramentas sob uma estrutura comum. Um exemplo é o ambiente fornecido pelo UNIX; outra concepção é a colocação de uma camada de controle sobre ferramentas já existentes uniformizando o uso das mesmas, e uma outra é fornecer um conjunto de serviços - pré-definidos - às ferramentas.

Um ambiente que pretenda servir como base de desenvolvimento de sistemas deve, se não agrupar todas as ferramentas necessárias às várias fases do projeto, fornecer meios de integrá-las posteriormente, e deve ter uma interface que possa se adaptar aos usuários em uma dada fase. No ambiente, as ferramentas devem ser fáceis de usar, devem ser uniformes e adaptáveis, segundo as características já discutidas.

Nos capítulos seguintes trataremos de LOTOS e de ferramentas que fazem parte de um ambiente de especificação de sistemas utilizando LOTOS.

Capítulo 3

LOTOS como linguagem de especificação de sistemas

3.1 LOTOS

LOTOS (Language of Temporal Ordering Specification) [ISO 87] é uma das linguagens de especificação formal desenvolvidas pela ISO para ser usada na especificação de sistemas abertos distribuídos. A idéia central de LOTOS é a descrição de sistemas a partir da relação temporal que existe entre as ações observáveis externamente em um sistema. LOTOS se baseia em *álgebra de processos*, um trabalho introduzido por Milner [Mil 79] [Mil 86] com CCS (Calculus of Communicating Systems) e CSP (Communicating Sequential Processes) [Hoa 85]. Complementarmente, LOTOS utiliza ACT ONE [Her 85] como base para descrição de estruturas de dados e expressões de valores.

LOTOS poderia ser aplicada para a descrição de qualquer sistema de informação concorrente e distribuído; entretanto, sua utilização principal é no âmbito dos sistemas abertos (OSI) definidos pela ISO [LSc 88][Vij 90][Sin 80].

A utilização de LOTOS como meio de descrição de sistemas OSI permite:

- a descrição precisa, completa e não ambígua de padrões
- a geração de documentos utilizados tanto por usuários como por implementadores e testadores

- a elaboração de bases formais para verificação e validação de padrões, bem como para os testes de conformidade de suas implementações [Bol 87] [Bri 86b].

A busca de uma linguagem formal de descrição pela ISO justifica-se pela ampla gama de utilização de seus padrões, onde não pode haver ambigüidades em definições, sob pena de chegar-se a implementações do mesmo padrão incompatíveis entre si. Outro fator ponderado é a possibilidade de se fazer verificação e testes das especificações a nível de projeto ("design") [Man 89].

Como linguagem formal de descrição de sistemas, LOTOS apresenta as seguintes características:

- poder de expressão
- definição formal, suportando o desenvolvimento de teorias analíticas para verificação, validação e conformidade
- abstração, não obrigando a expressão de detalhes de implementação na fase de projeto
- facilidades de estruturação, permitindo que uma especificação seja estruturada de forma a ser legível, fácil de manter, e permitindo análises.

Neste capítulo, apresentaremos os elementos básicos que formam a linguagem LOTOS. Através da descrição de LOTOS básico, descreveremos

apenas os métodos de sincronização entre processos, não havendo, nesta descrição, detalhes sobre a troca de informação entre processos. A descrição completa de LOTOS pode ser encontrada em [ISO 87] [Bol 87b]. Serão apresentados também os elementos básicos para descrição de tipos de dados em LOTOS

3.2 Conceitos fundamentais em LOTOS

3.2.1 Processos

Os sistemas distribuídos são descritos, em LOTOS, através de *processos*. Um processo é uma entidade que é capaz de se comunicar com o ambiente que a cerca, e que pode ter ações internas, não observáveis pelo ambiente. Um sistema como um todo pode ser visto como um processo, que pode ser formado de diversos sub-processos. Um sub-processo, por sua vez, é também um processo.

Deste modo, uma especificação de um sistema em LOTOS é feita através da elaboração de uma hierarquia de definições de processos.

Um processo pode ser visto como uma "caixa preta", em cuja especificação se definem as formas utilizadas por ela para interagir com o ambiente.

A interação de um processo com o ambiente é feita através de *eventos* ou *ações*. Um evento é uma unidade de comunicação sincronizada entre proces-

sos. Interações complexas entre processos são constituídas por vários eventos.

A definição de um processo em LOTOS especifica seu comportamento através da definição das ações do processo que podem ser observadas externamente. Utilizando o artifício de "caixa preta", podemos apresentar processos como na figura 3.1 .

Na figura 3.1, identificamos que os processos P e Q podem interagir através

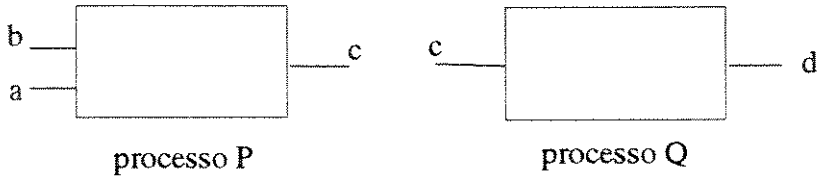


figura 3.1 - processos como caixas pretas

da porta "c". Apesar de vários eventos observáveis, não temos idéia de como seria a atividade interna dos processos.

O formato de uma definição de processo em LOTOS é:

process <identificador-processo><lista de parâmetros> :=
<Expressão de comportamento>
endproc

Nesta definição, observamos:

- um identificador, de forma a possibilitar que o processo possa ser referenciado
- uma expressão que define qual é o comportamento do processo que pode ser observado
- parâmetros, que representam a lista de eventos externos do processo. Os parâmetros também podem qualificar o tipo do processo[ISO 87].

3.2.2 Expressões de comportamento

Um componente essencial para a definição de um processo é a sua *expressão de comportamento* . Uma expressão de comportamento é constituída a partir de operadores básicos, que interligam várias expressões, e de instanciação de processos, através do uso de seu identificador.

A construção de expressões de comportamento é feita basicamente em termos de ofertas de interações. A interação é o mecanismo através do qual processos se comunicam ou se sincronizam. A interação se dá através de uma oferta de eventos em uma porta (gate). Se um processo oferece um evento em uma porta e outro processo oferece um evento na mesma porta, então pode ocorrer a interação.

Por exemplo, seja a expressão de um processo como segue segue:

p ?x:int ! true

Isto significa que o processo oferece uma interação na porta p, e são negociados dois valores, x do tipo int, sem valor conhecido, e o valor true.

Se um outro processo oferece:

p ! 3 ? y:bool

Haverá interação entre os processos, pois ambos referenciam a mesma porta *p* e há um "casamento" entre as informações passadas/solicitadas. Do resultado da interação teríamos "*x*" associado a "*3* " no primeiro processo e "*true*" associado a "*y*" no segundo processo. Nem sempre, entretanto, são necessários valores conhecidos para haver uma interação. LOTOS oferece ainda outros tipos de interações [Bol 87b].

Expressões de comportamento são utilizadas para expressar formalmente a ordem em que os eventos observáveis do processo podem ocorrer.

A tabela a seguir descreve os operadores básicos para construção de expressões de comportamento; para a descrição da sintaxe destes operadores, tomaremos B1 e B2 como expressões de comportamento quaisquer, e "*g*" como um evento qualquer.

nome	sintaxe
inativo	stop
ação observável	g;B_1
ação interna	*i*;B_1
escolha	B_1 [] B_2
composição paralela	
- caso geral	B_1 |[g_1, ... , g_2]| B_2
- entrelaçamento puro	B_1 ||| B_2
- sincronização completa	B_1 || B_2
ocultamento (hidding)	**hide** g_1, ... , g_n **in** B
instanciação de processo	P [g_1, ... , g_n]
terminação com sucesso	**exit**
composição sequencial	B_1 >> B_2
desabilitação	B_1 [> B_2

3.2.3 Operadores básicos

- Stop

  Stop representa, em LOTOS, um processo completamente inativo. Este processo não tem nenhum evento, e tem o mesmo papel, em LOTOS, do zero na aritmética.

- Ações

  Uma ação, representada por "g ;B ", onde g é um evento, descreve que o sistema comportará como a expressão B se o evento g ocorrer. Quando a ação é pré-fixada por "i", significa que esta ação é tomada internamente, e não é observada pelo ambiente. O operador ";" evidencia o caráter de seqüencialidade entre expressões de comportamento.

- Escolha (choice)
  O operador "B1 [] B2" representa que o comportamento do processo é descrito tanto pela expressão de comportamento B1 como pela expressão B2. A escolha entre uma ou outra é feita à medida em que o processo interage com seu ambiente. Se um evento relativo a B1 é oferecido, o sistema evoluirá conforme o comportamento descrito por B1; o mesmo acontece para B2. No entanto, se são fornecidos eventos tanto para B1 como para B2, a evolução é não determinística.

- Composição paralela
  Três tipos de representação para a composição paralela de processos são fornecidos em LOTOS. No caso em que dois processos têm comportamentos completamente disjuntos, usamos o operador "|||" para representar a relação entre as expressões de comportamento. Tomando os eventos observáveis e uma expressão "B_1 ||| B_2", podemos notar que, se um evento é aceito por B_1 ou por B_2, então a expressão como um todo participa do evento.
  No caso em que dois processos se comportam com dependência completa, ou seja, são compostos de forma paralela, porém com execução dependente um do outro, expressamos em LOTOS o comportamento através da expressão "B_1 || B_2". Desta forma, representamos a condição de que a expressão somente participa do comportamento do processo se B_1 e B_2 aceitarem o evento fornecido naquele instante.
  O caso geral de paralelismo de processos é representado por "B_1 |[a_1,...,a_n]| B_2", onde "a_1,...,a_n" são os eventos nos quais B_1 e B_2 devem se sincronizar.
  Podemos observar que:

  - o caso "||" representa o conjunto completo de eventos para sincronização entre B1 e B2

- o caso "|||" representa um conjunto vazio de eventos para a sincronização entre B1 e B2

- o caso "|[a_1,...,an]|" é o caso geral, onde os eventos para sincronização são expressos claramente.

. Hiding

O operador "hide" é usado nos casos onde uma ação é oculta do ambiente. A ação pré-fixada por "hide" é invisível ao ambiente, e ocorre sem ser observada e de forma espontânea, sem participação do ambiente. Podemos dizer que o operador "hide" introduz explicitamente ações não observáveis.

. Instanciação de processos

A instanciação de um processo "P [g_1, ... , g_n]" é formada pela referência ao identificador do processo ("P") e de uma lista de portas reais ("g_1, ... , g_n"). A instanciação assemelha-se a uma chamada de procedimento em linguagens de programação. Para ser instanciado, o processo deve ter sido definido na especificação. Através da instanciação de processos temos também a possibilidade da definição da recursão, representando assim comportamentos infinitos. Um processo recursivo instancia a si mesmo direta ou indiretamente.

. Terminação com sucesso (exit)

"Exit" é um operador cujo propósito é representar a terminação de um processo com sucesso. Este operador representa também seqüencialidade, pelo fato de habilitar a transferência do controle de um comportamento para outro quando usado em conjunto com o operador seqüencial.

. Composição seqüencial

Apesar de podermos expressar seqüencialidade através de

outros operadores (";", composição paralela onde a última ação de um processo coincide com a primeira ação do próximo), é desejável, por razões de clareza, ter um operador que represente explicitamente a composição seqüencial de processos. A idéia deste operador é evidenciar que um processo somente tem sua execução habilitada após a terminação, com sucesso, de um processo precedente.

Dada a expressão "B1 >> B2", interpretamos que, se B1 terminar com sucesso, então B2 poderá ser executado. Por este comportamento, este operador também é chamado de "enabling operator", ou operador de habilitação. Este operador pode ser usado em conjunto com a instanciação de processos de forma a representar partes do sistema independentemente, para depois instanciá-las em uma determinada seqüência.

- Desabilitação

  O operador "disabling" é utilizado para expressar uma ação de interrupção de serviço. Na descrição de protocolos OSI, acontecem casos onde a ação normal é interrompida a qualquer instante por eventos de desconexão ou aborto de conexão. Este operador é usado para representar estes casos. A expressão "B1 [> B2" define um processo onde, a qualquer ponto da execução de B1, pode ocorrer o evento inicial de B2. Se isto ocorrer, o controle das ações passa de B1 para B2. Se B1 termina antes desta ocorrência de evento para B2, então B2 não é executado. Se o evento de iniciação de B2 ocorre antes mesmo do evento inicial para B1, então B2 é executada.

3.3 Tipos de dados em LOTOS

LOTOS utiliza a linguagem de especificação de tipos abstratos de dados ACT ONE [Her 85] para descrever e representar valores, expressões de valores e estruturas de dados. Como linguagem de especificação de sistemas, LOTOS é consistente ao utilizar ACT ONE, uma vez que tipos abstratos de dados não devem levar em conta detalhes de implementação, ou seja, mantém-se a mesma abordagem utilizada para a definição de processos. Um tipo abstrato de dados age como um meio de especificação formal de classes de dados concretos, tais como conhecemos nas linguagens de programação convencionais. Para uma descrição de um tipo abstrato de dados, utilizamos conjunto de dados, e as operações que agem sobre os mesmos. A este conjunto, do ponto de vista matemático, dá-se o nome de "álgebra". Uma álgebra define a semântica, isto é, o significado de uma especificação. Os tipos abstratos de dados provêm modelos matemáticos para definir tipos de dados concretos; dados são tratados em termos de axiomas que governam o comportamento de estruturas de dados.

Através de tipos abstratos de dados (TAD) podemos definir as propriedades essenciais dos dados, bem como as operações que devem agir sobre aqueles dados sem, no entanto, explicitar como estas operações devem ser implementadas [Got 88] [Gar 89].

Para a produção de especificações de dados, LOTOS provê as seguintes facilidades:

- utilização de uma biblioteca de tipos de dados pré-definidos
- extensões e combinações de especificações já existentes
- renomeação de especificações
- parametrização de especificações
- substituição de parâmetros formais por parâmetros reais

A seguir, apresentaremos sucintamente os conceitos básicos que permitem oferecer as facilidades citadas acima. Estes conceitos são diretamente fornecidos por ACT ONE.

3.3.1 Conceitos de tipos de dados abstratos em LOTOS

- Assinatura

  Para definição de um tipo de dado abstrato devemos poder dar nomes aos conjuntos de dados e às operações sobre os mesmos. Chamamos de "sorts" os nomes dos conjuntos de dados. Para cada operação é definido um domínio que é constituído de zero ou mais "sorts", e uma imagem, ou resultado, que consiste exatamente de um "sort". É dado o nome de "assinatura" ao conjunto de "sorts" e operações de um tipo de dado. A assinatura de um tipo provê toda a informação necessária para construir *termos* sintaticamente corretos, ou expressões de valor, que representam valores de

dados (de algum "sort") daquele tipo.
ACT ONE provê regras gramaticais para produzir assinaturas compostas tanto de "sorts" como de operações e de combinações de ambos.

- Termos, equações e variáveis
  Os nomes dos "sorts" e as operações definidas em uma assinatura devem corresponder aos "sorts" e às operações de uma álgebra. Todos os elementos de um "sort" podem ser produzidos a partir das operações, se estas forem chamadas repetidamente. Um *termo* é o resultado desta repetição de operações sobre os elementos de um "sort". Cada termo representa um elemento de uma álgebra dada pela definição do conjunto "sort"+operações.
  Se utilizarmos outros operadores sobre um dado termo, iremos produzir outros termos. Para interpretar estes novos termos é necessária uma construção que expresse propriedades de operações. Esta construção é chamada de "equação".
  Para que uma equação seja expressa como válida para todos os elementos de um conjunto de dados é introduzida a noção de "variável". Uma variável representa a aplicação de uma operação ou equação a todos os elementos de um "sort".

- Combinações e extensões
  Para especificar tipos de dados com um grande número de operações é necessário que a linguagem permita compor e/ou estender especificações. Isto é feito acrescentando novos "sorts", operações e equações a uma especificação já existente. Combinar elementos permite que definamos uma especificação de forma incremental, iniciando-se da especificação de tipos de dados simples e utilizando-os como base para definições mais complexas.

- Parametrização

  A parametrização de uma especificação permite que se isole o princípio "gerador" do tipo abstrato de dados, de forma a que a especificação possa se aplicar a diferentes tipos de dados. A parametrização pode ocorrer quando este princípio "gerador" é independente dos objetos que o tipo abstrato de dados define. Exemplos desta característica são os tipos como pilhas e filas.

  Para se especificar um tipo abstrato de dados parametrizado isolamos as operações de forma a que elas ajam sobre um tipo formal. Este tipo formal pode conter equações formais, que são interpretadas como requisitos que um tipo de dados, passado como parâmetro, deve cumprir para ser aplicado àquela definição.

  Em suma, a parametrização é a combinação de uma parte que define formalmente um parâmetro e os "sorts", operações e equações que definem os tipos de dados dos parâmetros.

- Renomeação

  O conceito de renomeação é útil durante o desenvolvimento de uma especificação quando um tipo abstrato de dados, já definido, é necessário em um contexto diferente daquele para o qual foi originalmente definido. Neste caso, se não há alterações na semântica da definição, ACT ONE (e portanto LOTOS) permite que se defina um novo tipo a partir daquele já existente. Na definição por renomeação se definem um novo nome para o tipo e novos nomes para seus "sorts" e para suas operações.

A figura 3. mostra o exemplo de uma especificação de dados em LOTOS usando ACT ONE. Neste exemplo é mostrada a definição de um tipo de dado que representa os números naturais e as operações possíveis de se fazer sobre

eles. Note que além de definir as operações (cláusula **opns**), também é definida a semântica dessas operações (cláusula **eqns**)

3.4 Exemplo de uma especificação LOTOS

Na figura 3.2 é mostrado um exemplo de uma especificação LOTOS. Neste exemplo, são definidos dois processos. O processo *Buffer* oferece duas portas para sincronização : *"entra"* e *"sai"*. O comportamento desse processo é oferecer infinitamente os eventos *"entra"* e *"sai"* (notar a utilização da recursão) .

O segundo processo *Buffer_duplo* utiliza a definição do processo *"Buffer"* para especificar um comportamente onde são aceitas até duas entradas antes

```
specification  Buffer_duplo [entrada,saída] : exit
 behaviour
  hide meio in
       Buffer [entrada,meio]
         |[meio]|
       Buffer [meio,saída]
  where
   process Buffer[entra,sai] : exit :=
    entra ; sai ; Buffer[entra,sai]
   endproc
endspec
```

figura 3.2 - exemplo de uma especificação LOTOS

qua aconteça uma saída. O processo "*Buffer_duplo*" é definido como a composição paralela de dois processos "*Buffer*" . A figura 3. 3 ilustra de forma gráfica esta composição.

Note que é utilizado um evento interno definido explicitamente ("*meio*")

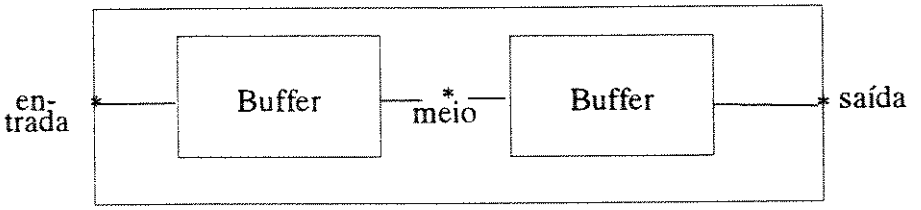


figura 3.3 - Composição de processos

para representar uma sincronização interna entre os processos sem a participação do ambiente. Exemplos de seqüências de eventos possíveis para o processo são:

1.entrada	2.entrada	3. entrada
entrada	saída	entrada
saída	entrada	saída
saída	saída	entrada
.....	.....	.....

Note que no máximo ocorrem dois eventos de entrada antes que seja obrigatório um evento de saída significando um buffer com capacidade 2.

```
type    BasicNaturalNumber is
sorts   Nat
opns    0                  : →Nat
        Succ               : →Nat
        _+_, _*_, _**_     : Nat, Nat →Nat

eqns    forall m,n: Nat
        ofsort Nat

        m+0                = m
        m+Succ(n)          = Succ(m) + n

        m*0                = 0
        m*Succ(n)          = m + (m*n)

        m**0               = Succ(0)
        m**Succ(n)         = m*(m**n)

endtype
```

figura 3.4 especificação de dado em LOTOS com ACT ONE

No capítulo seguinte, descreveremos o modelo de uma ferramenta integrada para LOTOS.

Capítulo 4

Modelo de uma ferramenta integrada para a linguagem LOTOS

Nos capítulos anteriores vimos que várias ferramentas são possíveis em cada fase do processo de desenvolvimento. Neste capítulo nos concentraremos naquelas ferramentes particularmente utilizadas na fase de especificação de um sistema utilizando LOTOS. Vimos que nesta fase o uso de ferramentas visa permitir :

- Construção e modificação do texto da especificação
- Verificação sintática
- Verificação semântica
- Verificação do comportamento dinâmico (simulação)

Descreveremos o modelo de uma ferramenta que possui a funcionalidade desejada acima. A estrutura deste modelo é mostrada na figura 4.0, sendo composta das seguintes partes:

- Editor : permite a criação e modificação de textos LOTOS.
- Analisador sintático : a análise sintática visa a verificação do texto LOTOS com as regras sintáticas definidas pela linguagem. O analisador sintático gera, para textos LOTOS sintaticamente corretos, uma representação alternativa chamada Árvore Sintática Abstrata (ASA) cujo objetivo é servir como representação básica para integração com outras ferramentas.

- Analisador semântico : a análise semântica visa a verificação do texto LOTOS sintaticamente correto com as regras da semântica estática definida para a linguagem. O analisador semântico utiliza a ASA como representação da especificação e a partir daí efetua análise semântica e produz uma nova estrutura denominada Árvore de Execução (AE). Esta nova estrutura é um refinamento da ASA, à qual são acrescidas informações semânticas.

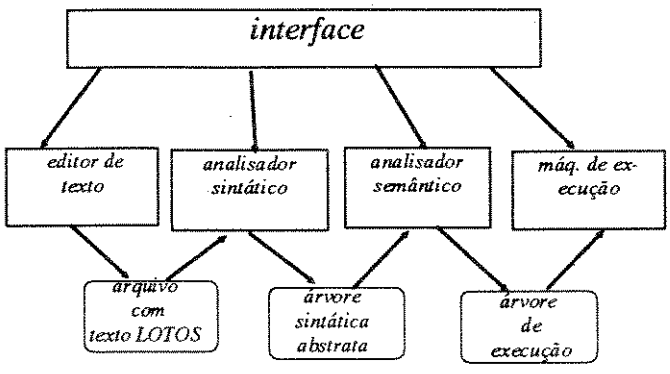


figura 4.0 - estrutura da ferramenta

- Máquina de Execução (MEL) : a máquina de execução LOTOS, ou seja, o núcleo do simulador, fornece funções básicas que possibilitam a animação da especificação. Essas funções permitem que se determine os próximos eventos possíveis e que se execute um passo de simulação a partir de um determinado evento.

- Interface com usuário : a interface com o usúario define as possibilidades de interação entre o usuário e a ferramenta.

Nas seções seguintes detalharemos cada um destes componentes com ênfase nos aspectos referentes à simulação. A descrição do editor e interface com

usuário será feita de maneira informal enquanto para os demais componentes será utilizada uma forma mais elaborada, de maneira a fornecer uma descrição construtiva (algoritmica) que permita derivar facilmente uma implementação.

Exemplos de abordagens diferentes de ferramentas para LOTOS podem ser encontrados em [Log 88] [Eij 89b] [Ghl 88] [Que 88] [Boc89].

4.1 O Editor de Texto

Um dos propósitos da ferramenta descrita neste trabalho é o fornecimento de um ambiente integrado para a elaboração de especificações usando LOTOS. Neste sentido, entendemos que o usuário, sob um mesmo ambiente, deve poder escrever sua especificação, corrigi-la automaticamente e depois simulá-la. A primeira interface apresentada ao usuário é a do editor; as outras operações podem ser indiretamente acessadas através de comandos oferecidos pelo editor.

O editor, por si próprio, é uma ferramenta que permite a construção, modificação e inspeção dos "objetos" que compõem a especificação [Eij 89]. Editores operam sobre seqüências de caracteres ou linhas e podem, ou não, impor uma estruturação do texto que é criado/modificado pelo usuário. Esta estruturação, em geral, está ligada à própria estrutura do produto final do qual

o editor proporciona a execução - uma especificação, um texto qualquer, um programa, uma carta etc.

Algumas funções mínimas devem ser oferecidas pelo editor que compõe a ferramenta e que são funções "usuais" de editores: inserção e eliminação de caracteres, busca e troca de caracteres ou seqüência de caracteres, apresentação do texto editado. Outras funções suplementares como menus para apresentação de comandos ou múltiplas janelas de edição também são desejáveis.

O fato do editor ser o elemento do ambiente onde se concentra a interação com o usuário o torna um ponto de integração para as atividades de outras ferramentas. Ou seja, sendo possível ativar, dentro do editor, as ações de compilação e simulação, o ambiente se torna mais homogêneo do ponto de vista do usuário, que não precisa aprender e utilizar outras interfaces para atingir seu propósito final, que é o desenvolvimento de uma especificação. Para que isto aconteça é necessário que o ambiente sobre o qual as ferramentas operam mantenha ou suporte o compartilhamento de informações entre as ferramentas. Por exemplo, erros referentes à análise sintática resultantes da operação do compilador (i.e, do analisador sintático) podem ser mostrados na tela do editor, bem como as evoluções possíveis da simulação, resultantes da operação do simulador.

4.2 O Simulador

Um simulador de uma especificação é a parte de uma ferramenta que permite a análise do comportamento dinâmico descrito pela especificação. Em geral, esta análise se dá *passo a passo*, havendo interação com o projetista a cada "decisão" de comportamento. Um simulador deveria, basicamente, computar um conjunto de eventos aceitáveis a partir de uma especificação. Estes eventos seriam então "oferecidos" ao projetista (ou usuário da ferramenta), que poderia escolher um evento; a partir desta escolha, o simulador calcularia o novo comportamento resultante. Este procedimento seria repetido sucessivas vezes, sob controle do usuário[BFL 86] [Bri 86] [Tre 89] [Pap].

Podemos observar, da definição anterior, que as funções mínimas que um simulador deveria oferecer ao usuário são: uma lista de eventos a serem escolhidos, o estado no momento da escolha, o estado após a escolha de um daqueles eventos e uma forma de *evoluir* o comportamento da especificação a partir das opções selecionadas pelo usuário. A esta categoria de funções, onde essencialmente se "percorre" a especificação por meio de seleção de eventos e observação dos estados resultantes, chamaremos de "funções de navegação" [Eij 89]. Nesta categoria, além das funções mínimas citadas, englobamos todas as operações que permitam a "visita" a todas as "partes" de uma especificação. Tomando como modelo básico o fato de uma especificação poder ser expressa em uma árvore [1] [Gll 88], isto significa que as

funções de navegação deveriam permitir a visita a todos os nós desta árvore (que representam os estados possíveis de uma simulação), oferecendo como meio para isso os ramos, que são os eventos possíveis a partir de um estado (nó).

Além das funções de navegação, outras operações poderiam ser feitas sobre a árvore a título de simulação. Além da animação do comportamento descrito pela especificação, o simulador poderia apresentar funções que desempenham papel de verificador e validador da mesma, ou gerar casos de testes, ou mesmo apresentar um protótipo do comportamento real do sistema descrito. Nesta concepção, estas outras operações seriam encarregadas de colocar os nós ou estados da especificação sob testes de verificação ou, sob um sentido mais amplo, informar quanto da especificação foi "exercitada" dentro de uma simulação (e portanto quão completa foi esta simulação). De uma maneira geral, podemos agrupar as funções de um simulador da seguinte maneira:

1. Funções de navegação

As funções básicas de navegação, como já citado anteriormente, são:

1 dada pela semântica de LOTOS

- apresentação das opções de eventos possíveis, dado um estado (menu de opções). Isto significa, no modelo de árvore comunicante, a apresentar o conjunto de caminhos possíveis a partir de um nó.

- permitir a passagem para um estado subseqüente a partir de um estado atual e um evento.

- mostrar as característica do estado em que se encontra (em qualquer etapa do processo de simulação)

Além destas funções, um simulador poderia apresentar ainda operações para:

- retornar a um estado anteriormente visitado

- mostrar o caminho percorrido até o estado atual

- mostrar os caminhos possíveis (sub-árvores) a partir do estado atual

- aceitar um conjunto pré-definido de eventos e simular automaticamente o comportamento da especificação a partir deste conjunto

- alterar o caminho de execução para outra parte da especificação ainda não visitada ("jump")

Estas operações poderiam fazer parte de uma futura evolução do trabalho desta tese.

2. Funções de validação e análise de cobertura

Funções de validação têm como atributo básico a verificação e testes de um nó ou estado. A verificação das propriedades de um estado pode detectar problemas como deadlock, livelock, determinar equivalências a outros comportamentos apresentados na especificação. Podemos dizer que quaisquer propriedades que possam ser descritas formalmente poderiam ser analisadas por funções desta categoria.

Já as funções de cobertura têm como responsabilidade determinar quanto de uma especificação foi "coberta" em uma simulação. Em outras palavras, isto significa saber quanto da árvore que representa a especificação foi explorada, e portanto, determinar quão completa foi esta exploração. Para isto, o simulador poderia registrar a quantidade de estados visitados, quantos caminhos da especificação poderiam ser exercitados e, em cooperação com as funções de validação, determinar caminhos equivalentes, diminuindo portanto os caminhos necessários para esgotar as possibilidades de simulação.

4.3 Análise sintática: geração da árvore sintática abstrata

O processo de análise sintática resulta na validação das construções sintáticas do texto LOTOS verificando sua conformidade com a gramática da linguagem definida em [ISO 87] . Com a análise sintática, a forma textual da

- Analisador semântico : a análise semântica visa a verificação do texto LOTOS sintaticamente correto com as regras da semântica estática definida para a linguagem. O analisador semântico utiliza a ASA como representação da especificação e a partir daí efetua análise semântica e produz uma nova estrutura denominada Árvore de Execução (AE). Esta nova estrutura é um refinamento da ASA, à qual são acrescidas informações semânticas.

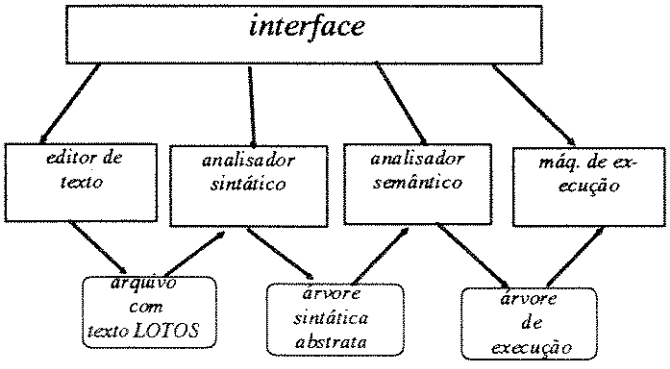


figura 4.0 - estrutura da ferramenta

- Máquina de Execução (MEL) : a máquina de execução LOTOS, ou seja, o núcleo do simulador, fornece funções básicas que possibilitam a animação da especificação. Essas funções permitem que se determine os próximos eventos possíveis e que se execute um passo de simulação a partir de um determinado evento.

- Interface com usuário : a interface com o usúario define as possibilidades de interação entre o usuário e a ferramenta.

Nas seções seguintes detalharemos cada um destes componentes com ênfase nos aspectos referentes à simulação. A descrição do editor e interface com

usuário será feita de maneira informal enquanto para os demais componentes será utilizada uma forma mais elaborada, de maneira a fornecer uma descrição construtiva (algoritmica) que permita derivar facilmente uma implementação.

Exemplos de abordagens diferentes de ferramentas para LOTOS podem ser encontrados em [Log 88] [Eij 89b] [Ghl 88] [Que 88] [Boc89].

4.1 O Editor de Texto

Um dos propósitos da ferramenta descrita neste trabalho é o fornecimento de um ambiente integrado para a elaboração de especificações usando LOTOS. Neste sentido, entendemos que o usuário, sob um mesmo ambiente, deve poder escrever sua especificação, corrigi-la automaticamente e depois simulá-la. A primeira interface apresentada ao usuário é a do editor; as outras operações podem ser indiretamente acessadas através de comandos oferecidos pelo editor.

O editor, por si próprio, é uma ferramenta que permite a construção, modificação e inspeção dos "objetos" que compõem a especificação [Eij 89]. Editores operam sobre seqüências de caracteres ou linhas e podem, ou não, impor uma estruturação do texto que é criado/modificado pelo usuário. Esta estruturação, em geral, está ligada à própria estrutura do produto final do qual

o editor proporciona a execução - uma especificação, um texto qualquer, um programa, uma carta etc.

Algumas funções mínimas devem ser oferecidas pelo editor que compõe a ferramenta e que são funções "usuais" de editores: inserção e eliminação de caracteres, busca e troca de caracteres ou seqüência de caracteres, apresentação do texto editado. Outras funções suplementares como menus para apresentação de comandos ou múltiplas janelas de edição também são desejáveis.

O fato do editor ser o elemento do ambiente onde se concentra a interação com o usuário o torna um ponto de integração para as atividades de outras ferramentas. Ou seja, sendo possível ativar, dentro do editor, as ações de compilação e simulação, o ambiente se torna mais homogêneo do ponto de vista do usuário, que não precisa aprender e utilizar outras interfaces para atingir seu propósito final, que é o desenvolvimento de uma especificação. Para que isto aconteça é necessário que o ambiente sobre o qual as ferramentas operam mantenha ou suporte o compartilhamento de informações entre as ferramentas. Por exemplo, erros referentes à análise sintática resultantes da operação do compilador (i.e, do analisador sintático) podem ser mostrados na tela do editor, bem como as evoluções possíveis da simulação, resultantes da operação do simulador.

4.2 O Simulador

Um simulador de uma especificação é a parte de uma ferramenta que permite a análise do comportamento dinâmico descrito pela especificação. Em geral, esta análise se dá *passo a passo*, havendo interação com o projetista a cada "decisão" de comportamento. Um simulador deveria, basicamente, computar um conjunto de eventos aceitáveis a partir de uma especificação. Estes eventos seriam então "oferecidos" ao projetista (ou usuário da ferramenta), que poderia escolher um evento; a partir desta escolha, o simulador calcularia o novo comportamento resultante. Este procedimento seria repetido sucessivas vezes, sob controle do usuário[BFL 86] [Bri 86] [Tre 89] [Pap].

Podemos observar, da definição anterior, que as funções mínimas que um simulador deveria oferecer ao usuário são: uma lista de eventos a serem escolhidos, o estado no momento da escolha, o estado após a escolha de um daqueles eventos e uma forma de *evoluir* o comportamento da especificação a partir das opções selecionadas pelo usuário. A esta categoria de funções, onde essencialmente se "percorre" a especificação por meio de seleção de eventos e observação dos estados resultantes, chamaremos de "funções de navegação" [Eij 89]. Nesta categoria, além das funções mínimas citadas, englobamos todas as operações que permitam a "visita" a todas as "partes" de uma especificação. Tomando como modelo básico o fato de uma especificação poder ser expressa em uma árvore [1] [Gll 88], isto significa que as

funções de navegação deveriam permitir a visita a todos os nós desta árvore (que representam os estados possíveis de uma simulação), oferecendo como meio para isso os ramos, que são os eventos possíveis a partir de um estado (nó).

Além das funções de navegação, outras operações poderiam ser feitas sobre a árvore a título de simulação. Além da animação do comportamento descrito pela especificação, o simulador poderia apresentar funções que desempenham papel de verificador e validador da mesma, ou gerar casos de testes, ou mesmo apresentar um protótipo do comportamento real do sistema descrito. Nesta concepção, estas outras operações seriam encarregadas de colocar os nós ou estados da especificação sob testes de verificação ou, sob um sentido mais amplo, informar quanto da especificação foi "exercitada" dentro de uma simulação (e portanto quão completa foi esta simulação). De uma maneira geral, podemos agrupar as funções de um simulador da seguinte maneira:

1. Funções de navegação

As funções básicas de navegação, como já citado anteriormente, são:

1 dada pela semântica de LOTOS

- apresentação das opções de eventos possíveis, dado um estado (menu de opções). Isto significa, no modelo de árvore comunicante, a apresentar o conjunto de caminhos possíveis a partir de um nó.

- permitir a passagem para um estado subseqüente a partir de um estado atual e um evento.

- mostrar as característica do estado em que se encontra (em qualquer etapa do processo de simulação)

Além destas funções, um simulador poderia apresentar ainda operações para:

- retornar a um estado anteriormente visitado

- mostrar o caminho percorrido até o estado atual

- mostrar os caminhos possíveis (sub-árvores) a partir do estado atual

- aceitar um conjunto pré-definido de eventos e simular automaticamente o comportamento da especificação a partir deste conjunto

- alterar o caminho de execução para outra parte da especificação ainda não visitada ("jump")

Estas operações poderiam fazer parte de uma futura evolução do trabalho desta tese.

2. Funções de validação e análise de cobertura

Funções de validação têm como atributo básico a verificação e testes de um nó ou estado. A verificação das propriedades de um estado pode detectar problemas como deadlock, livelock, determinar equivalências a outros comportamentos apresentados na especificação. Podemos dizer que quaisquer propriedades que possam ser descritas formalmente poderiam ser analisadas por funções desta categoria.

Já as funções de cobertura têm como responsabilidade determinar quanto de uma especificação foi "coberta" em uma simulação. Em outras palavras, isto significa saber quanto da árvore que representa a especificação foi explorada, e portanto, determinar quão completa foi esta exploração. Para isto, o simulador poderia registrar a quantidade de estados visitados, quantos caminhos da especificação poderiam ser exercitados e, em cooperação com as funções de validação, determinar caminhos equivalentes, diminuindo portanto os caminhos necessários para esgotar as possibilidades de simulação.

4.3 Análise sintática: geração da árvore sintática abstrata

O processo de análise sintática resulta na validação das construções sintáticas do texto LOTOS verificando sua conformidade com a gramática da linguagem definida em [ISO 87] . Com a análise sintática, a forma textual da

especificação LOTOS é transformada numa árvore denominada *Árvore Sintática Abstrata (ASA). A ASA* é uma representação alternativa para uma especificação LOTOS e é obtida através de uma função de mapeamento:

- *G-ASA: LOTOS TEXTUAL* →ASA

O domínio desta função são as construções sintáticas LOTOS. Uma *ASA* é uma árvore rotulada cujos nós consistem de um indicador do elemento sintático que o nó representa e atributos, cujos ramos são zero ou mais *ASAs*. Os nós da *ASA* estão relacionados às produções da gramática concreta da linguagem. Durante a análise sintática os identificadores definidos no texto de uma especificação são armazenados numa estrutura de dados chamada *contexto estático* . No que concerne à análise sintática, o contexto estático é representado por um conjunto de identificadores estendidos . Um identificador estendido é o identificador acrescido de informações de contexto. A geração deste contexto não é importante para construção da ASA (uma vez que a ASA é apenas uma representação sintática alternativa) mas será útil na fase posterior durante a validação semântica. A ocorrência de identificadores na representação textual é separada em duas classes:

- definição: o identificador ocorre durante uma definição
- uso: o identificador ocorre numa referência de uso

A ocorrência de definição de um identificador implica na geração do identificador estendido e sua conseqüente introdução no contexto estático. A ocorrência de uso de um identificador não provoca alteração de contexto estático, gerando apenas uma referência ao *contexto estático* no nó da *ASA*. onde o identificador é usado.

4.3.1 A sintaxe abstrata

A importância de se definir uma sintaxe abstrata no que diz respeito a uma linguagem é hoje uma prática largamente reconhecida e consolidada. Esta prática surgiu da consideração de que uma especificação ou programa (em um sentido mais geral) escrito numa linguagem não deve ser considerado apenas como uma seqüência de caracteres ou linhas de texto mas como um conjunto de objetos estruturados. Esses objetos são formados por elementos que são significativos do ponto de vista da semântica da linguagem. As regras (ou objetos) que definem a estrutura da linguagem formam a sintaxe abstrata da linguagem. É importante ressaltar que uma descrição no nível de sintaxe abstrata elimina as seguintes dificuldades :

- uma representação particular desses objetos
- regras de precedência e/ou associatividade de operadores
- ambigüidades na interpretação de uma representação particular de uma construção

Tais tipos de problemas ocorrem apenas quando tratamos com representações sintáticas concretas. Outro ponto interessante é que representações concretas, bem como qualquer outra interpretação semântica, podem ser obtidas a partir de um mapeamento da sintaxe abstrata para o domínio de interesse. Uma vez definida uma sintaxe abstrata, esta não necessita ser alterada se são definidas novas sintaxes concretas. A sintaxe abstrata apresenta-se, portanto, como elemento central em qualquer ferramenta que precise manipular um texto da especificação, especialmente num ambiente integrado de desenvolvimento, onde temos normalmente diversas ferramentas integradas, cada uma enfocando a especificação dentro do seu domínio específico.

Intuitivamente, a definição de uma sintaxe abstrata pode ser obtida através das seguintes regras [Hee 90]:

1. definir um conjunto de categorias sintáticas da linguagem. Essas categorias agrupam construções homogêneas. Exemplo de grupo sintático em LOTOS é a Expressão de Comportamento, que agrupa construções homogêneas como Expressão Paralela, Expressão Seqüencial.

2. definir cada construção em termos de outras construções (sub-construções)

3. definir construções básicas que não incluem outras sub-construções

Como se pode ver, uma vez efetuado os passos acima teremos uma estrutura hierárquica em forma de árvore com as construções básicas como folhas. Definiremos uma sintaxe abstrata para LOTOS segundo os passos acima.

De uma maneira geral, a sintaxe abstrata para LOTOS que utilizaremos é definida como a linguagem gerada pela gramática cujas regras de produção têm a seguinte forma:

```
categoria sintática :== construção 1
                      | construção 2
                      :
                      construção n
construção :==        nome da construção
                      ( sub-construção 1 :    categoria sintática
                       sub-construção 2  :    categoria sintática
                       :
                       sub-construção n  :    categoria sintática
                      )
                      | construção básica
construção básica :== elemento básico
```

Uma categoria sintática pode admitir vários tipos de construção. O símbolo | separa contruções alternativas para uma determinada categoria sintática. Por exemplo, a categoria sintática relativa a expressões de comportamento

admite uma construção distinta para cada tipo de expressão (expressão paralela, hide, etc.). Uma construção (uma instância específica de uma categoria sintática) é identificada por um nome único chamado *rótulo da construção*. As sub-construções associadas a uma construção são também categorias sintáticas e portanto podem ser instanciadas com qualquer construção que pertença a esta categoria. Para permitir representar uma construção genericamente sem determinar suas sub-construções , são atribuídos nomes a estas sub-construções. Esses nomes funcionam como variáveis que podem conter qualquer construção (do tipo adequado) como valor. Dessa maneira podemos ter representações parciais de uma construção sem expandir suas sub-construções. Chamaremos de *representação completa* uma construção sem variáveis. Uma representação de uma construção com variáveis será chamada de incompleta. As construções que não possuem sub-construções são chamadas construções básicas e representam as folhas da árvore. Um exemplo de categoria sintática básica são os identificadores que não possuem nenhuma subconstrução. No Anexo 4 temos a gramática da sintaxe abstrata para LOTOS conforme utilizaremos na construção do ambiente integrado.

Definimos a *ASA* como uma representação completa da especificação num formato que satisfaz a gramática da sintaxe abstrata. A característica principal da *ASA* é a ausência de elementos sintáticos "explícitos". O que caracteriza sintaxes concretas são, por exemplo, o que se chama comumente de "palavras

chaves". Na gramática concreta de LOTOS a palavra *specification* é uma "palavra chave".

As construções sintáticas válidas para a linguagem LOTOS são descritas pela gramática concreta da linguagem, que é expressa em [ISO 87] em BNF. Nesta gramática, o elemento não terminal specification é (os elementos em negrito são palavras-chave, enquanto entre <> são não terminais):

<specification> ::= **specification** <spec-id> <formal-parameters>
behaviour
<definition-block>
endspec

Da mesma forma, na sintaxe abstrata podemos definir um nó que representa a mesma construção:

A-Specification ::= *SpecDef (1)*
(SpecId: A-Ident
FormalGates: A-IdentList
FormalValues: A-IdentDeclarations
Functionality: A-Exit
GlobalType: A-GlobalTypeDef
Block: A-DefinitionBlock
)

Na sintaxe abstrata desaparecem as palavras chaves, os identificadores são atributos dos nós e os símbolos não terminais são também representados em

sintaxe abstrata. Os elementos não terminais da sintaxe são precedidos de "A" (Abstrato) na representação abstrata de um nó.

4.3.2 A função G-ASA

A função G-ASA (função geradora da ASA) é uma função que transforma um texto LOTOS que satisfaz os requisitos sintáticos da sintaxe concreta da linguagem numa representação completa ASA. A função é definida recursivamente sobre os *não-terminais* da gramática concreta de LOTOS . Dessa maneira o domínio da função G-ASA consiste de todo texto gerado a partir desses *não-terminais* . A imagem da função G-ASA são todas as ASA's geradas a partir da gramática que define a sintaxe abstrata, ou seja :

- G-ASA : Texto Lotos → ASA

A definição da função G-ASA é obtida através de um mapeamento entre não terminais da gramática concreta em construções da sintaxe abstrata. Intuitivamente, a idéia é que, dado um texto que satisfaz uma determinada regra de produção da gramática concreta, digamos R_i, mapeamos este texto a partir dos elementos que compõem a regra em um nó da ASA. Tomemos por exemplo um texto que satisfaz a regra R_1:

- Process-Instantiation :==<Process-Ident><Actual-Gate-List>

Geramos então com G-ASA(R_1,) um nó da forma:

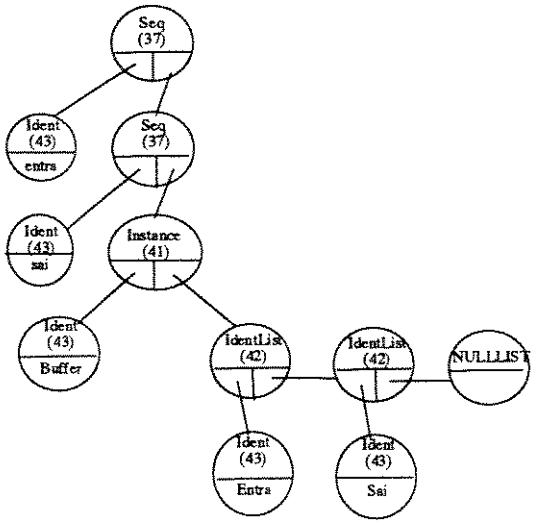


figura 4.1 - árvore sintática abstrata

- *Instantiation*(G-ASA(Process-Ident),G-ASA(Actual-Gate-List))

Note que as sub-construções do nó *Instantiation* são obtidas também utilizando recursivamente a função G-ASA aplicada aos sub-componentes da regra de produção que originou o nó. A expressão de comportamento do processo *Buffer,* definido no capítulo anterior, produz a árvore mostrada graficamente na figura 4.1, que condiz com o especificado no anexo 4.

4.4 Análise semântica: a Árvore de Execução

A análise semântica é feita a partir de um texto LOTOS sintaticamente correto, transformado pelo processo de análise sintática em uma ASA (

Árvore Sintática Abstrata). O resultado da análise semântica é a validação da ASA e do contexto estático CE formado pelas definições de nomes e a criação de uma nova estrutura denominada "Árvore de Execução" (AE). A fase de análise semântica portanto engloba:

1. verificação dos requisitos semânticos, conforme descrito em [ISO 87] no que diz respeito ao de controle.

2. transformação do texto LOTOS (representado pela ASA equivalente) em uma nova estrutura mais adequada ao processo de simulação, chamada "Árvore de Execução" (AE).

3. validação do contexto estático CE contendo todos os nomes declarados na especificação. Durante a análise semântica os nomes são acrescidos de informações adicionais referentes ao escopo onde eles foram definidos.

A Árvore de Execução está relacionada à expressão de comportamento LOTOS no que diz respeito à parte de controle. A geração da AE e a validação do CE é definida através da função *Check*. Esta função define o mapeamento de um texto LOTOS em um contexto estático válido e uma AE. A função *Check* está definida apenas para textos LOTOS semanticamente corretos, isto é, para textos LOTOS que satisfazem os requisitos de semântica estática conforme [ISO 87].

4.4.1 Estrutura do contexto estático

O contexto estático é constituído de vários conjuntos onde cada conjunto define uma classe de identificadores. Quando um identificador é inserido em um determinado conjunto, são agregadas informações adicionais. Essas informações dependem da classe a que ele pertence. O identificador com as informações adicionais é chamado de *identificador estendido* . Uma informação importante que é associada ao identificador é o escopo. Um escopo define um ambiente para o qual um dado identificador é visível, isto é, neste ambiente este identificador pode ser referenciado. Por exemplo, os parâmetros formais (gates) definidos para um processo podem ser referenciados apenas dentro da expressão de comportamento do processo. Se chamarmos esta expressão de EC, definimos o escopo de um gate G_i do processo como:

- escopo(G_i) = scp(EC)

Dentro de um determinado escopo podem ser definidos novos identificadores que consequentemente terão um ambiente onde podem ser referenciados mais restrito que o escopo externo. Por exemplo, se a EC for uma composição paralela da forma:

- $EC = EC_1 \parallel EC_2$, e
- EC_2 = hide g_1,g_2 in EC_2' , então
- escopo(g_1)=scp(EC_2')

Neste exemplo, o escopo do gate interno g_1 está contido no escopo de EC_2'. Usaremos os símbolos '=' e $\subset$ para definir respectivamente as relações de igualdade e inclusão entre escopos. Neste mesmo exemplo temos que escopo(g_1)=escopo(g_2) e scp(EC_1) $\subset$ scp(EC).

Os conjuntos, as classes e as respectivas informações adicionais (que formam o contexto estático) são os seguintes:

- NS: conjunto contendo identificadores "Sort-estendido" de classe SORT. Um sort estendido[2] é definido como :
  Sort-estendido = < id-sort,tipo-estendido,scp>
  - id-sort = identificador do sort,
  - tipo-estendido = referência ao tipo (estendido) ao qual este sort está relacionado,
  - scp = escopo no qual o sort foi definido.
- NO: conjunto contendo identificadores "Operador-estendido" da classe OPERATOR. Um operador estendido é definido como:

  Operador-estendido = <id-op,tipo-estendido,args,res,pos,scp >
  - id-op = identificador do operador.

2 Os *identificadores estendidos* são escritos como uma lista da forma <id,a_1,...,a_n > onde id é a sequência de caracters do identificador e a_k ($1 \le k \le n$) são informações adicionais.

- tipo-estendido = referência ao tipo ao qual este operador está relacionado.

- args = lista dos sorts dos argumentos do operador na forma $<s_1,...,s_n>$ onde s_k $(1 \leq k \leq n)$ é um sort estendido. Para operadores sem argumentos (constantes) esta lista é vazia.

- res = sort estendido que representa o sort do resultado da operação.

- pos = denota a posição do operador com relação aos argumentos: infixo ou prefixo. No caso de operadores sem argumentos, esta informação não é considerada.

- scp = escopo.

• NT: conjunto contendo identificadores "Tipo-estendido" da classe TYPE. Um tipo estendido é definido como:
Tipo-estendido = <id-tipo,import-tipo,scp>

- id-tipo = identificador do tipo.

- Import-tipo = lista dos tipos importados por este tipo na forma $< t_1, ..., t_n >$ onde t_k $(1 \leq k \leq n)$ é um tipo estendido. Para tipos sem importação esta lista é vazia.

- scp = escopo.

• NP: conjunto contendo identificadores "Processo-estendido" da classe PROCESS. Um processo estendido é definido como :
Processo-estendido = <id-proc,glist,vlist,func,scp,btree>

- id-proc = identificador do processo.

- glist = lista dos gates formais do processo na forma $<g1,...,gn>$ onde g_k $(1 \leq k \leq n)$ é um gate estendido.

- vlist = lista das variaveis (valores) formais do processo na forma $<v_1,...,v_n>$ onde vk $(1 \leq k \leq n)$ é uma variável estendida. Para processos sem variáveis como parâmetros formais esta lista é vazia.

- func = lista dos sorts que representam a funcionalidade do processo na forma $<s_1,...,s_n>$ onde s_k $(1 \leq k \leq n)$ é um sort estendido. Para processos com funcionalidade EXIT ou NO-EXIT esta lista contém respectivamente o símbolo especial ExitFunc ou NoExitFunc.

- scp = escopo.

- btree = é a arvore gerada pelo expressão de comportamento que representa o processo.

- NG: conjunto contendo identificadores "Gate-estendido" da classe GATE. Um gate estendido é definido como:
  Gate-estendido = <id-gate,scp>

  - id-gate = identificador do gate.

  - scp = escopo.

- NV: conjunto contendo identificadores da classe VARIABLE. Uma variável estendida é definida como:
  Variável-estendida =<vid,scp,vtree>

  - vid = identificador da variável.

  - scp = escopo.

- vtree = representação em árvore do valor assinalado à variável. este valor é um termo aberto ACT ONE.

Para o exemplo do capítulo 3 figura 3.2 , o conjunto NG de gates do contexto estático é o seguinte:

NG={ <*entrada*,sc_1>,<*saída*,sc_1>,<*meio*,sc_2>,<*entra*,sc_3>,<*sai*,sc_3>}

Os símbolos *entrada* e *saída* pertencem ao mesmo escopo sc_1. Da mesma forma, os símbolos *entra* e *sai* pertencem ao mesmo escopo sc_3 . O símbolo *meio* ainda que definido dentro do mesmo processo que os símbolos *entrada* e *saída* tem escopo diferente sc_2 restrito à expressão "*hide*".

4.4.2 Árvore de execução

A semântica operacional de LOTOS [ISO 87] fornece um meio de se derivar sistematicamente as ações que um processo pode oferecer a partir da própria estrutura da expressão de comportamento. De uma maneira mais precisa, dada uma expressão de comportamento EC, derivamos *transições rotuladas* que são tuplas da forma:

$$EC \xrightarrow{a} EC'$$

Onde *a* é uma ação e EC' é outra expressão de comportamento. Da mesma forma na simulação teremos que derivar todas as ações possíveis para um

processo a partir da representação da expressão de comportamento. Já temos uma representação sintática alternativa para a especificação LOTOS (ASA). Definimos agora outra representação chamada *Árvore de Execução* (apenas para as expressões de comportamento) , que será utilizada pelo simulador para determinar quais as ações que um processo pode oferecer. As características da AE são :

- Somente um texto LOTOS que satisfaça os requisitos de sintaxe e semântica estática da linguagem tem uma representação em AE.
- Cada nó da árvore de execução contém uma instrução derivada de um operador LOTOS.
- As definições de processo possuem uma AE que pode ser referenciada através do nome do processo.

Neste texto representaremos um nó da AE como uma lista da forma :

$<rótulo, <p_1>, ..., <p_n> >$

onde o rótulo é a instrução derivada do operador LOTOS e os parâmetros da instrução são objetos sobre os quais a instrução opera. Exemplos de parâmetros são nomes de gates, processos ou outros nós (representando sub-expressões compostas)

A transformação da Expressão de Comportamento para a Árvore de Execução é feita usando as regras de derivação baseadas nos operadores LOTOS *Op*. Para o operador de prefixação ";" a regra é a seguinte:

- se Op é um operador ";" de prefixação com :
  EC= $g;EC_1$
  deriva-se a árvore

  $<Seq, g, AE_1>$

O Anexo 1 contém as regras para os demais operadores.

A diferença principal entre a ASA e a Árvore de Execução é que nesta última as referências a identificadores são substituídas por referências a identificadores estendidos que são os identificadores acrescidos de informação de contexto. O objetivo do identificador estendido é, além de associar informações que permitem a verificação da semântica estática, incluir também informações que auxiliarão no processo de simulação. Através dos identificadores estendidos é possível distinguir entre dois identificadores que têm a mesma grafia. A distinção é feita através do escopo, de maneira que existe apenas um identificador estendido com um determinado escopo e grafia. A definição do identificador estendido equivale à planificação (*Flattenning*) dos identificadores conforme definido em [ISO 87].

A árvore de execução para a expressão de comportamento do processo *Buffer* descrito no exemplo do capítulo 3 (figura 3.2), é semelhante a ASA da figura 4.1 a menos dos identificadores que são acrescidos de informação de contexto (escopo), conforme mostrado na figura 4. 2.

4.4.3 Análise Semântica

Definiremos na seqüência a análise semântica feita sobre os nós da ASA. Para cada nó são definidas as ações tomadas que se refletem na geração da Árvore de Execução ou alteração do contexto estático (representado pela inserção dos identificadores estendidos nos conjuntos NP,NG) e também são verificados se as informações no nó atendem a requisitos estáticos estabelecidos (por exemplo, um gate referenciado num nó do tipo *seq* tem que estar definido no contexto estático). Como não estaremos tratando a parte ACT ONE, a informação contida nos nós relativos à parte de dados não será considerada. Chamaremos a função de análise semântica de *Check* . A cada aplicação da função *Check* será sempre considerada a existência do contexto estático representado pelos conjuntos NP,NG e da informação de um escopo corrente (os conjuntos NT,NV,NO,NS,NV não serão considerados pois não estamos tratando a parte relativa a dados). A aplicação da função em um nó provoca uma alteração no contexto estático sempre que um identificador é definido e provoca uma consulta sempre que um identificador é referenciado. Durante a análise semântica a função *Check* também define

a *funcionalidade* das expressões de comportamento e é verificada a sua concordância com a funcionalidade explicitada na definição do processo correspondente. A *funcionalidade* de uma expressão de comportamente pode assumir os seguintes valores:

- EXITFUNC
- NOEXITFUNC
- INDEFINIDA

As funcionalidades EXITFUNC e NOEXITFUNC referem-se à terminação com sucesso ou terminação com insucesso respectivamente. A funcionalidade INDEFINIDA refere-se a uma expressão composta mal formada, isto é, que não está de acordo com requisitos semânticos de composição de expressões no que diz respeito à funcionalidade. Para verificar composição de expressões usaremos as seguintes funções auxiliares:

$Min(func_1,func_2)$ =	NOEXITFUNC	se $func_1$=NOEXITFUNC ou se $func_2$ = NOEXITFUNC
	$func_1$	se $func_1$=$func_2$
	INDEFINIDA	se diferente das anteriores

$Max(func_1,func_2)$ =	func1	se $func_2$=NOEXITFUNC
	func2	se $func_1$=NOEXITFUNC
	func1	se $func_1=func_2$
	INDEFINDA	se diferente das anteriores

A função *Check* para o nó *Seq* é a seguinte :

- Check(*Seq*)(SCP)
  g=CheckGate(A-Ident)(SCP)
  AE=<*Seq*,g,AE_1>
  AE_1=Check(A-BehaviourExpress)(SCP)
  funcionalidade(Seq)=funcionalidade(AE_1)
  Notas:
  - (a) A função CheckGate deve retornar um identificador estendido $g \in NG$ com escopo(g)=SCP do contrário temos um erro semântico.

Para os demais nós a definição da função Check é feita no Anexo 2.

4.5 O núcleo do simulador LOTOS

Descreveremos agora o núcleo do simulador que chamaremos de MEL : Máquina de Execução LOTOS. A MEL é uma máquina que fornece uma funcionalidade mínima exigida para a simulação da linguagem e que consiste basicamente de duas funções:

- Uma função que determina os próximos eventos possíveis para evolução da simulação.

- Uma função que efetua uma evolução da simulação a partir de um determinado evento possível.

Definiremos a MEL como uma máquina virtual que possui um conjunto de operadores semanticamente equivalentes aos operadores da linguagem LOTOS. No que diz respeito à simulação da linguagem LOTOS podemos considerar um simulador como constituído de duas partes distintas: um executor de expressões de comportamento relativo à parte de controle e um avaliador de expressões ACT ONE relativo à parte de dados. Esses dois aspectos são ortogonais [Led 87] podem ser tratados isoladamente. A intersecção entre estas duas partes se dá durante a evolução (passo de execução) em sincronizações com passagem de valor , na análise de predicados, e durante o cálculo dos eventos em expressões que contêm guarda. Em [Kbe 70] é mostrado um algoritmo que soluciona uma classe de problemas semelhante ao encontrado na implementação de um avaliador de expressões ACT ONE . Como não estamos tratando a parte de LOTOS relativa a dados, não desenvolveremos nenhuma idéia adicional com relação ao avaliador de expressões e simplesmente não o levaremos em consideração para o simulador proposto. Somente a parte de controle da linguagem será considerada. Cabe considerar que as ferramentas aqui desenvolvidas operam

dentro do conceito de LOTOS básico, ja'que não incluem a parte relativa à definição de dados (ACT ONE).

4.5.1 As funções básicas da MEL

A definição da semântica dinâmica da linguagem LOTOS é feita em termos de regras de inferências. Do ponto de vista de simulação a especificação das funções do simulador usando estas regras traz a desvantagem de não ser construtiva (ou algorítmica) no sentido em que ela não pode ser mapeada diretamente para uma linguagem de programação convencional. De uma certa maneira é possível se utilizar desta característica para transformar estas regras em programas em linguagens lógicas [Gil 89] [Gil] [Log 85] [Sid 83] ou funcionais [Mdm 88]. Em [BFL 86] é mostrado um esquema em que as regras são traduzidas em um programa Prolog que gera todas as transições possíveis. Em [Eij 86] é feita uma comparação de simuladores de linguagens comportamentais implementado com linguagens de programação lógica.

Descreveremos a funcionalidade da MEL a partir da Árvore de Execução de maneira a permitir um mapeamento direto para linguagens de programação imperativas. Para isto utilizaremos as regras de inferência definidas para uma dada expressão e a partir da AE equivalente a esta expressão transformamos a regra em uma definição algorítmica. Vamos analisar inicialmente o significado de uma expressão de comportamento LOTOS , considerando a semân-

tica dinâmica da linguagem. De acordo com a semântica LOTOS, dada uma expressão de comportamento EC temos o seguinte:

(1) se EC -a → EC', dizemos que o comportamento será descrito por EC' na ocorrência do evento a. Isto é, na ocorrência do evento a , a expressão evolui para EC'.

(2) se EC -a_1 → EC_1 ,...,EC -a_n → EC_n , dizemos que o comportamento na ocorrência de um evento a_i ($1 \leq i \leq n$) será dado pela expressão EC_i .

Vemos que se coletarmos todos os eventos a_i para os quais uma expressão de comportamento pode evoluir temos a funcionalidade exigida que determina os eventos possíveis. Da mesma forma se obtivermos a expressão EC_i resultante da ocorrência do evento a_i temos a funcionalidade exigida para se evoluir a simulação (passo de execução). Definiremos a MEL como constituída de duas funções básicas, sendo uma que coleta as opções de eventos possíveis que chamaremos de *Options* e outra que efetua a evolução da simulação obtendo a próxima expressão resultante da ocorrência de um determinado evento. Chamaremos esta função de *Next* . Mais precisamente temos:

- *Options* : AE → G, onde AE= Árvore de execução e G é um conjunto de oferta de eventos.
- *Next* : G x AE → Bool x AE .

A função *Options* é uma função que tem como argumento uma Árvore de Execução e retorna um conjunto de gates onde eventos são ofertados. A função *Next* tem como argumento uma árvore de execução e um evento e retorna uma tupla formada de um valor booleano e uma árvore de execução. A idéia é que a partir de uma AE equivalente a uma dada expressão de comportamento EC tenhamos :

- se EC - $a_1 \rightarrow EC_1$,..., EC - $a_i \rightarrow EC_i$,
  então Options(AE) = G, onde G={a_1,...,ai }.

Da mesma forma:

- se EC - $a_n \rightarrow EC_n$,
  então Next(AE,a_n) = <True,AE_n > , onde o valor True indica que o passo foi executado e AE_n é a árvore de execução que representa EC_n .

Precisamos agora determinar o comportamento das funções *Options* e *Next* para cada uma das instruções MEL. Faremos isto sempre utilizando uma AE e a expressão de comportamento relativa a esta árvore, de modo a estabelecer uma equivalência funcional (de maneira bastante informal) entre a MEL e LOTOS através da análise comparativa da expressão de comportamento LOTOS e da operação da MEL na árvore equivalente. A partir da semântica

do operador LOTOS expresso na EC, determinaremos a operação da máquina sobre a AE através das funções *Options* e *Next*.

4.5.2 Semântica de MEL : a função *Options*

Como já vimos no item anterior a função *Options* deve colecionar todos os eventos possíveis em um dado instante da simulação. Chamaremos de estado uma AE que representa uma dada expressão de comportamento. Como em cada estado estaremos sempre com um nó da AE , definiremos a função *Options* para cada nó possível. A definição é recursiva no sentido em que o resultado da análise de um nó pode ser obtido através da aplicação da própria função em ramos que derivam do nó em questão. Uma observação importante é que estaremos considerando cada evento como um elemento independente de maneira que a oferta de dois eventos num mesmo gate seja diferenciado, ou seja, os eventos a_i e a_j com $i \neq j$ podem referenciar o mesmo gate mas ser elementos independentes dentro do conjunto de ofertas possíveis. Trataremos indistintamente o evento e o gate onde o evento é ofertado, desde que isto não traga confusão. Neste caso apontaremos a diferença. O significado da função *Options* para o nó *Seq* (nó equivalente ao operador sequencial ";") é o seguinte :

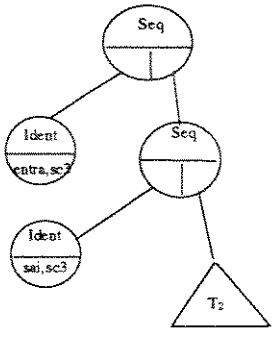
- se EC=g;EC1, AE=<Seq,g,AE1> ,
  então

Options(AE)= {g} , onde g é o gate do referenciado no nó *Seq*

A definição da função *Options* para os demais nós é feita no Anexo 3.

4.5.3 A função Next

Uma vez definido o método através do qual se coleta os próximos eventos, veremos agora como podemos calcular o novo estado (representado por uma nova AE) a partir de um estado presente e um dado evento. O cálculo deste novo estado, é determinado pela função *Next* , é equivalente a um passo de simulação. Descreveremos a função do mesmo modo que fizemos no item anterior, ou seja, definiremos a funcionalidade a partir dos nós da Árvore de Execução. Para o nó *Seq* a definição é a seguinte:



árvore de execução equivalente à expressao EC=entra;sai;T_2

figura 4.2 - árvore de execução

1. AE=<*Seq*,g,AE_1>
   se a=evento(g) então
    Next (a,AE) = <True,AE_1>
   senão
    Next(a,AE) = <False,AE>

Nota : evento(g) significa o evento ofertado no gate g .

Para os demais nós a definição da função é feita no Anexo 3.

Considerando o exemplo do capítulo 3 (figura 3.2), a árvore de execução referente à expressão de comportamento do processo ***Buffer*** é a seguinte :

T= <*Seq*,<entra,sc_3>,T_1>

onde T_1 é a sub-árvore :

T_1 =<*Seq*,<sai,sc3>, T_2>

ou graficamente como mostra a figura 4.2.

A função *Options* aplicada a árvore T (consequentemente ao nó *Seq*), retorna conjunto de eventos EV={ <entra> } com um único elemento o evento <entra> . A função *Next* aplicada T para este evento retorna <True, T_1 >.

No capítulo seguinte descreveremos uma implementação deste modelo utilizando uma linguagem de programação imperativa.

Capítulo 5

Implementação do modelo

Neste capítulo descreveremos a implementação de um programa que fornece a funcionalidade descrita para uma ferramenta integrada LOTOS conforme o capítulo anterior. A estrutura da implementação reflete a estrutura funcional no sentido que para cada unidade funcional existe uma unidade de implementação. Dessa maneira, cada unidade de implementação pôde ser desenvolvida isoladamente sendo a comunicação entre as unidades feita através da base de dados. Todo o software da ferramenta foi implementado nas linguagens C e C++, a menos do editor de texto, onde foi utilizado um programa cujo fonte é disponível comercialmente. Tal editor é fornecido na linguagem de programação PASCAL e faz parte de um conjunto de ferramentas conhecida como TURBO TOOLBOX [TURB].

5.1 O editor de texto

Editores de texto são ferramentas utilizadas em nosso dia a dia constantemente. Existem diversos tipos, com as mais diferentes funcionalidades . A adição de um editor de texto à ferramenta descrita neste trabalho foi orientada no sentido de prover uma interface agradável e auxílio à construção de especificações LOTOS. A utilização de uma ferramenta pronta - o Turbo Editor Toolbox - facilitou o trabalho de incorporação do editor à ferramenta LOTOS. Sua escolha foi feita baseando-se no fato de ser o único disponível como ferramenta integrável ao resto do código, e por apresentar as características de "customização" como veremos a seguir.

O Turbo Editor Toolbox é um editor de textos comercial disponível em forma de um programa fonte pascal. O programa é extremante modular permitindo a criação de editores particulares para uma determinada aplicação. O Turbo

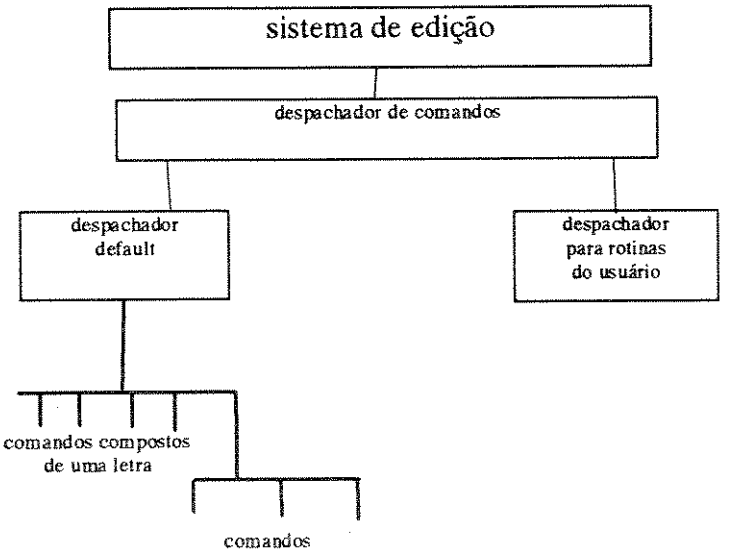


figura 5.1 - estrutura do editor

Editor permite edição de arquivos de qualquer tamanho, sendo que a estrutura do "buffer" de texto de edição que mantem (ou seja, a forma como armazena os caracteres que compõem o texto editado) é de uma lista ligada de registros Pascal, cada um deles contendo apontadores para:

- a linha anterior
- a linha posterior

- a cadeia de caracteres que representa uma linha no texto

Esta estrutura é usada por permitir alterações com facilidade, além de ser possível gerenciar qualquer número de linhas. Para compensar o acesso freqüente do disco para armazenar o "buffer" de edição (um acesso a cada linha), o Turbo Editor contém rotinas especiais de acesso de alto desempenho.

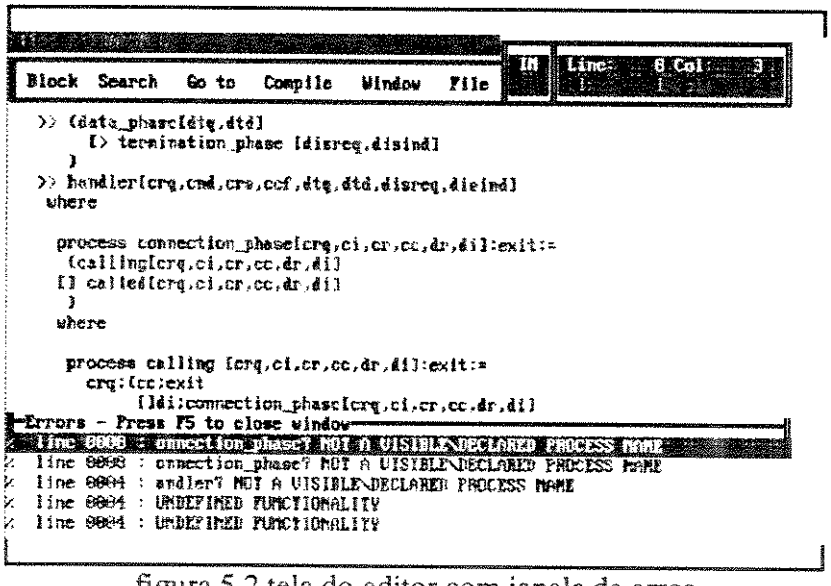


figura 5.2 tela do editor com janela de erros

O Turbo Editor apresenta uma estruturação como descrito na figura 5.1 .

Dada esta estruturação, a "customização" do Turbo Editor consiste, primeiramente, em se construir um programa que:

- chame uma rotina pré-definida de inicialização interna de estruturas de dados

- passe o controle para o sistema de edição do Turbo Editor
- no fim dos trabalhos, ''limpe" a tela de edição

Em seguida, deve ser elaborado o tratamento dos comandos próprios do ambiente que se implementa. No caso da ferramenta LOTOS, são reconhecidas teclas especiais como por exemplo:

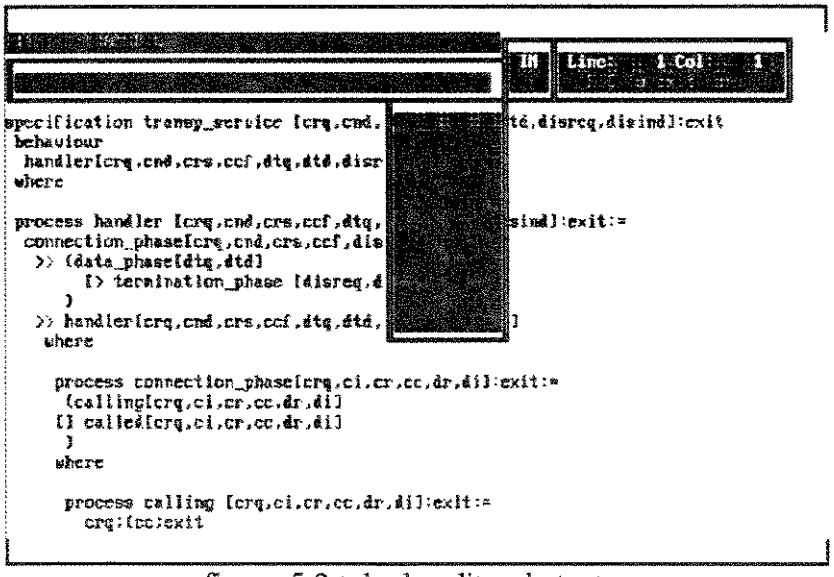


figura 5.3 tela do editor de textos

- [F9]: ativa o compilador LOTOS para efetuar análise sintática e semântica da especificação LOTOS editada no ambiente de edição. No caso de erros, utilizamos um mecanismo de múltiplas janelas, provido pelo editor, para permitir a visualização dos erros. Criamos um modo de ''correção de erros" para o editor, no qual os erros

podem ser ''varridos" seqüencialmente [1]; a cada erro, o cursor é posicionado na linha do texto fonte onde está o erro.

- [F10]: sai do modo ''edição" e entra em um modo de simulação, onde é executado o simulador LOTOS

Como base do editor utilizamos um código fonte com diversas facilidades, como menus, janelas, e tratamento de erros customizado.

A figura 5.2 mostra uma tela do editor. A janela superior contém o texto sendo editado e a janela inferior contém os erros da compilação do mesmo texto. A figura 5.3 mostra uma tela do editor contendo apenas a janela de edição e com o menu de tratamento de arquivos ativado.

5.2 Analisador sintático e semântico

Durante a descrição das funções de análise sintática e semântica da ferramenta vimos que são referenciadas vários conjuntos que representam o domínio ou imagem das funções descritas. Cada conjunto descrito possui uma estrutura de dados correspondente na implementação. As estruturas principais que são referenciadas são as seguintes :

- O contexto estático

1 também através de teclas especiais

- A Árvore Sintática abstrata

O contexto estático contém os identificadores declarados na especificação, tendo sido portanto implementado como uma tabela de símbolos. O mapeamento do contexto estático para uma tabela de símbolos foi feito segundo os seguintes passos:

- o identificador é codificado através de uma função de hashing em um número inteiro.
- O identificador é representado como uma estrutura **C** com a seguinte declaração:

```
struct Symbol {
    int code;
    int class;
    int *scope;
    Symbol * next;
    union {
        EType TypeId;
        EOper OperationId;
        EProc ProcessId;
        ESort SortId;
        EGate GateId;
        EValue ValueId;
    } S
}
```

Onde o campo "code" é o código do identificador, ''class" é a

classe do identificador, ''scope" é escopo do identificador e os diversos membros da union representam as informações particulares de cada uma das classes possíveis do identificador. Cada classe também é uma estrutura C. A estrutura Symbol representa o identificador estendido. Dois identificadores com mesma grafia possuem o mesmo código mas são diferenciados pelas demais informações e são agrupados como uma lista ligada (campo *next* da estrutura) .

- A tabela de símbolos foi representada como um vetor com acesso hashing onde o índice da tabela é o código do símbolo.

- Os caracteres que representam o identificador são armazenados apenas uma vez.

Note que o identificador com um elemento sintático é representado pelo seu código enquanto o identificador estendido (elemento semântico) é representado pela estrutura Symbol .

Para representar a Árvore Sintática Abstrata é feita uma estrutura C que reflete a mesma estrutura dos nós definidos para a ASA no Anexo 4 . Para cada nó é definida uma estrutura C e todas estas estruturas são compostas num union que representa um nó genérico chamado TreeNode. A estrutura TreeNode é codificada da seguinte maneira:

```
struct TreeNode {
    Tlabel label;
```

```
    union {
      ASpec SpecDef;
     AGlobalType GlobalDef ;
      ADefBlock  DefBlock;
      ALocalProc ProcDef;
      ALocalData DataDef;
      ALibDef    LibDef;
      . . .
      AIdentEqn IdentEqn;
     }
  }
```

O campo "label" identifica o tipo do nó e os membros da union descrevem um nó em particular (respectivo ao valor do label).

5.3 A geração da ASA usando LEX e YACC

Na construção de compiladores é uma prática comum estabelecer uma distinção entre a árvore de parse (do inglês parse tree) de uma linguagem qualquer L e sua árvore sintática abstrata. A árvore de parse é uma derivação que mostra como um texto em questão pode ser derivado de um símbolo inicial da gramática L [Hee 90]. Os nós internos de uma árvore de parse são os não terminais da gramática enquanto suas folhas são tokens léxicos como palavras chaves, identificadores, etc. A sintaxe abstrata, como vimos no capítulo 4, contém apenas informações essenciais mantendo a semântica da

linguagem. Geradores automáticos de parser (do inglês parser generators) são ferramentas que auxiliam na geração de programas que efetuam análise sintática, permitindo a tradução da árvore de parse para a sintaxe abstrata [Aho 77]. Exemplos dessas ferramentas são o Metal e o conjunto Lex/Yacc [Les 78] [Joh 78]. O compilador Metal traduz especificações Metal para um gerador de parser e analisador léxico do sistema Mentor. O Lex/Yacc são, respectivamente, o gerador de analisador léxico e o gerador de parser do sistema Unix . Esses últimos têm sido largamente utilizado e também já se encontram disponíveis para ourtros sistemas. Uma descrição mais abrangente pode ser encontrada em [Axe 85].

Lex usa expressões regulares para descrever a sintaxe dos tokens léxicos. Estes são compilados em tabelas para um autômato finito determinístico. Uma definição típica LEX para identificadores é a seguinte :

```
[a-z][a-z0-9]*  { slookup(yytext); }
```

A parte da esquerda desta regra descreve a sintaxe do identificador (que é : um identificador é uma letra seguido de zero ou mais letras ou dígitos) e a parte da direita escrita entre { } descreve a ação a ser tomada quando a sequência de caracteres de entrada casa com a regra especificada. As ações são escritas na linguagem C. No exemplo acima, quando o analisador léxico encontra um identificador é chamada a função *slookup*. A variável yytext é

uma variável global predefinada pelo Lex e é parte da interface entre o analisador léxico gerado e as ações escritas pelo usuário.

Yacc usa uma notação BNF para definir a gramática da linguagem para a qual se deseja gerar o parser e compila essas regras para tabelas de um parserdo tipo shift/reduce. O Yacc exige que a gramática satisfaça as restrições LALR(1). A sintaxe de entrada para o Yacc gerar o reconhecimento de uma ocorrência sintaticamente correta de uma expressão atômica é :

```
Atomic_express  : STOP
                    { $$=GenerateStop(); }
                | EXIT
                    { $$=GenerateExit( NULL); }
                | EXIT Exit_par_list
                    { $$=GenerateExit( $2); }
                | Process_instanc
                | OPENPAR Behavi_express CLOSEPAR
                    { $$=$2; }          ;
```

Da mesma forma que o lex, a ação a ser executada quando é feita uma "redução" (uma construção é identificada), é especificada como código C entre {}. No exemplo acima, as palavras totalmente em maiúsculo são palavras chaves da sintaxe concreta. As outras palavras são não terminais (e também tem uma regra própria) . Ainda neste mesmo exemplo, quando uma expressão atômica é identificada, é gerado um nó da ASA. Note que para a mesma construção existe mais de uma regra de produção separada por "|".

Para diferentes regras de produção são executadas diferentes ações. Quando uma redução é equivalente a um nó na ASA, este nó é gerado. Neste exemplo são gerados os nós *Exit* e *Stop*.

5.4 O núcleo do simulador

As estruturas de dados para o analisador sintático e semântico são descritas em linguagem *C* pura. As demais estruturas são mais elaboradas e são descritas em *C++* [Stroup]. A razão de se utilizar *C++* somente para representar o conjunto dos eventos e a Árvore de execução é que a geração da tabela de símbolo e da ASA são feitas com auxílio de um gerador de parser (que gera código C puro). Além disso, o compilador *C++* não estava disponível no início da implementação. A utilização de *C++* após a implementação de parte da ferramenta não apresentou nenhum problema, visto que *C++* é um "superset" de *C*. A Árvore de execução e o Conjunto de Eventos são implementados como classes em *C++* e portanto englobam tanto dados como funções.

A estrutura básica em que o simulador LOTOS se baseia é a Árvore de Execução. Sobre ela estão definidas as funções *Options* e *Next*, que são funções do simulador. Conforme descrito no capítulo 4 , a semântica LOTOS é definida em termos de *nós* de uma árvore. Cada nó (equivalente a um

operador LOTOS) é definido como um objeto que encapsula um comportamento próprio. A funcionalidade do nó é definida em termos das funções *next* e *options* que representam sua *interface.*

É interessante considerar que a abordagem de simuladores para LOTOS, em geral, se baseia no fato de LOTOS ter sua semântica definida em termos de regras de derivação e, com isso, tem seu enfoque nestas regras para sua implementação. É o caso de simuladores desenvolvidos em linguagem com suporte à definição de regras de derivação (ex: PROLOG). No caso deste trabalho, o enfoque foi dado nos operadores da linguagem, ou seja, a transformação semântica de LOTOS foi elaborada a partir de algoritmos relativos aos operadores. Um operador pode ser visto como um elemento de encapsulamento, ao qual são fornecidos valores. Em linguagens orientadas a objetos, o elemento de encapsulamento é uma classe. Portanto, a elaboração de classes que mapeiam os operadores LOTOS, tantos quanto são estes operadores, foi o ponto principal para a implementação deste simulador LOTOS utilizando os princípios de orientação à objeto. Cada operador de LOTOS aparece como uma classe; na transformação da especificação para simulação, cada operador da especificação forma um nó da árvore de execução. Desta maneira, a representação da AE como uma classe em C++ aparece como uma transformação bastante natural, visto que uma classe C++ possui um comportamento explicitado através de métodos que constituem sua interface.

A propósito, nota-se uma tendência a uma visão "orientada a objeto" da linguagem LOTOS. Em [Kor 89] [Lus 89] [Bla 89] [May 88] são abordados assuntos relativos a uma interpretação de LOTOS segundo o paradigma de orientação por objeto.

Os nós da AE não são homogêneos, isto é, cada nó possui uma representação própria e possui uma semântica própria para as funções *Next* e *Options*. Para tornar o tratamento destas funções homogêneo (independentes do nó) utilizamos os mecanismos de derivação de classe e funções virtuais de *C++*. Esses mecanismos permitem que, dada uma classe básica com funções virtuais $f_1, f_2, \ldots, f_n$, e classes derivadas com as mesmas funções, a função realmente chamada durante a execução seja a definida para a classe derivada. Todo este processo é transparente, ficando a cargo do compilador.

Para o simulador LOTOS definimos uma classe básica chamada ExecTree e uma classe derivada para cada nó particular da Árvore. A definição da classe básica ExecTree é a seguinte:

```
class ExecTree: {
 public:
  virtual EventSet Options();
  virtual ExecTree * Next (Event) ;
  virtual void Print();
 }
```

A classe ExecTree serve apenas como ponte para a definição de todas as outras classes derivadas. Vamos analisar, em detalhes, duas classes derivadas relativas aos nós *Seq* e *Choice* (o processo de derivação das demais classes é semelhante). É importante observar que a identificação do nó (rótulo) definido na especificação da AE serve apenas para diferenciar cada nó em particular. Na implementação em *C++* esta identificação fica implícita no nome da própria classe. A definição da classe relativa ao nó *Seq* é a seguinte:

```
class SeqNode: ExecTree {
 Symbol Gate;
 ExecTree * Tree;
  public:
    EventSet Options();
    ExecTree * Next( Event e);
    void Print();
}
```

A classe "SeqNode" tem como atributos o "gate" e uma AE que representa o comportamento seguinte (em sequência). Note a consistência da definição da classe com a definição do nó e a relação direta entre os dados do nó (representado pelo gate e a árvore seguinte) com os atributos da classe e as funções especificadas. A função "Options", segundo a definição, deve retornar o "gate" relativo ao nó, sendo portanto implementada como:

```
EventSet * SeqNode :: Options() {
 Event * Ev = new Event;
  Ev - >gate = Gate;
 EventSet * Set = new EventSet ( Ev );
 return * Set;
}
```

A função "Options" para o nó *Seq* retorna um conjunto de eventos cujo único elemento é o evento ofertado no gate referenciado no nó. Observe que a função "Options" retorna um conjunto e não um evento. A classe EventSet referenciada na definição da ''SeqNode" é a representação C++ do conjunto de eventos. A função "Next", conforme definida na especificação, deve retornar uma tupla composta de um valor booleano e uma AE. Na implementação da função *next*, usando o fato de que a AE só faz sentido quando o valor booleano retornado é TRUE, ao invés de retornar uma tupla (o valor booleano e a AE) retornamos apenas o valor da AE. Usamos o valor NULL para indicar retorno de um valor booleano FALSE (e portanto a árvore não faz sentido), ou a própria árvore, significando também um valor booleano TRUE. A definição da função "Next" para a classe ''SeqNode" é a seguinte:

```
ExecTree * SeqNode :: Next (Event) {
 if ( Event == Ev )
 return ( Tree ) ;
else
 return NULL;
}
```

Da mesma forma, para o nó *Choice* temos a seguinte definição:

```
class ChoiceNode :: ExecTree {
 ExecTree * LeftTree;
 ExecTree * RightTree;
 EventSet Set1;
 EventSet Set2 ;
  public:
   EventSet Options();
   ExecTree * Next (Event e) ;
   void print();
};
```

A classe "ChoiceNode" tem como dado duas sub-árvores correspondentes às sub-árvores definidas na especificação do nó Choice. Além disso, a classe possui dois conjuntos "Set1" e "Set2", que serão usados pela função "Next". Esse conjuntos não estão definidos na especificação do nó mas representam otimizações na implementação de "Next". A função "Options" para o nó "ChoiceNode" é a seguinte:

```
EventSet ChoiceNode :: Options() {
 EventSet S1 = LeftTree.Options();
 EventSet S2 = RightTree.Options();
  EventSet * S3 = S1 + S2 ;
 Set1.join(S1);
 Set2.join(S2);
 delete S1; delete S2;
 return S3;
}
```

A função "Options" para o nó "ChoiceNode" apresenta algumas características interessantes. Inicialmente são calculados os conjuntos de eventos de cada sub-árvore. Aqui vemos claramente a vantagem do uso de classe derivada. Uma vez que as sub-árvores LeftTree e RightTree são da classe ExecTree e portanto podem ser instanciadas com objetos de qualquer classe derivada, a função "Options" pode ser usada recursivamente sem necessidade do conhecimento prévio de qual é o nó atribuído a essas variáveis. No cálculo das opções totais para o nó, que é a união dos eventos possíveis para cada sub-árvore, temos mais uma facilidade de *C++*: a possibilidade de redefinição de operadores. O operador "+" da expressão S1 + S2 representa a união entre conjuntos, que é o que desejamos. A função "Next" para o nó "ChoiceNode" é a seguinte:

```
ExecTree * ChoiceNode ::Next(Event ev) {
 if ( Set1.isMember ( ev ) )
  return ( LeftTree.Next( ev ) ) ;
 else
  return ( RightTree.Next ( ev ) ) ;
 }
```

Vimos aqui que utilizamos os conjuntos "Set1" e "Set2" para decidir a priori se o evento a ser avaliado pertence à sub-árvore esquerda (LeftTree.Next(ev)) ou então o evento deve ser avaliado pela sub-árvore direita (RightTree.Next(ev)). De acordo com a definição de "Next" para o nó *Seq*,

deveríamos ter:

```
ExecTree * ChoiceNode:: Next (Ev) {
  if ( ( ExecTree * E1 = LeftTree(Ev).Next(Ev) != NULL) )

   return E1 ;
  else
   return ( RightTree(Ev).Next(Ev) );
  }
```

A opção pela implementação anterior é apenas uma questão de otimização de implementação, pois nesta segunda forma sempre teríamos que avaliar a sub-árvore esquerda mesmo que o evento se referisse à sub-árvore direita.

A definição das funções "Options" e "Next" para as classes que representam os demais nós da árvore seguem um raciocínio análogo.

Outra função importante do simulador é a função "print". Ainda que não definida explicitamente no capítulo anterior, devemos de alguma maneira poder exibir ao usuário o estado atual da simulação. Como o estado é representado pela árvore de execução corrente, definimos a função "print" como a função que imprime a árvore num formato adequado. Da mesma forma que as funções "Next" e "Options", a informação a ser impressa depende do nó e portanto é definida como uma função virtual na classe base

ExecTree e é definida em cada classe derivada. Para o nó "SeqNode" a função é definida da seguinte maneira:

```
void SeqNode:: Print() {
 if ( curcollumn  NMAXCOL - size (Gate) - 1 ) {
  curcollumn = 1;
  cout < NEWLINE;
  printGate ( Gate );
  }
else {
  printGate(Gate);
 cout < ";"
  curcollumn ++;
 Tree.Print();
  }
}
```

Aqui, mais uma vez, é utilizada recursão na chamada de "Tree.Print()". Para o nó "ChoiceNode", "Print" é definida como:

```
void ChoiceNode:: Print() {
 cout < "(";
 int cursor =  curcollumn ;
  LeftTree.Print();
  cout < NEWLINE;
  curcolumn = cursor;
  RightTree.Print();
  cout < NEWLINE;
  curcolumn = cursor;
```

```
    cout < ")";
}
```

Na definição das classes anteriores, fizemos referências ao conjunto de eventos. Este conjunto foi implementado como uma classe derivada de uma classe básica "slist". Esta classe é descrita como exemplo em [Stroup] e fornece funções básicas para manipulação de listas. A classe que representa os eventos é definida da seguinte maneira:

```
class EventSet :: public slist {
public:
void Print();
void insert ( Event * a ) { slist :: insert (a); }
void append ( Event * a ) { slist :: append (a); }
void join ( EventSet * s ) { slist :: join ( (slist *) s ); }
int isMember ( Event * e ) { slist :: isElement (e); }
EventSet Operator + (EventSet &E1, EventSet &E2) ;
EventSet() { }
EventSet( Event * a ) : slist (a) { }
};
```

A classe EventSet efetua apenas conversão de tipos, fornecendo uma nova interface para utilização da classe "slist". As funções "insert", "append", etc. estão descritas em [Stroup].

Capítulo 6

Conclusões

Ferramentas auxiliam no processo de desenvolvimento se elas possibilitam ou facilitam uma determinada atividade dentro do processo. Neste trabalho, descrevemos a implementação de um ambiente integrado para desenvolvimento de sistema usando LOTOS. O principal objetivo deste ambiente é o auxilio à fase de especificação do sistema. O ambiente fornece os seguintes serviços:

- editor de texto
- analisador sintático e semântico
- simulador

Através das ferramentas acima o ambiente permite a execução das seguintes atividades de forma totalmente integrada:

- edição do texto que compõe a especificação
- validação da sintaxe e semântica da especificação
- prototipação do sistema especificado, possibilitando uma verificação do comportamento do sistema ainda na fase de especificação.
- validação de seqüências de teste também através de prototipação

Uma característica particular deste ambiente reside no fato do simulador ter sido escrito numa linguagem orientada para objetos. O uso de C++ como

linguagem de implementação possibilita uma interpretação para os operados LOTOS usando o conceito objeto. Outra característica interessante reside no fato das ferramentas terem sido desenvolvidas para ambiente IBM PC, o que permite uma penetração maior dentro da comunidade acadêmica tendo em vista a grande difusão que este tipo de equipamento tem dentro do meio.

A escolha de LOTOS como linguagem de descrição de sistema foi feita baseado no grau de formalismo que a mesma impõe, juntamente à capacidade de expressão de problemas complexos como os que encontramos em ambientes distribuídos.

Acreditamos que este trabalho vem contribuir para a difusão da linguagem LOTOS e do uso de TDF em geral [Bri 86a].

Como continuidade do trabalho sugerimos uma ampliação da funcionalidade do ambiente com o adicionamento de novos componentes como formatadores, inclusão de tipos abstratos de dados no simulador, etc. A representação interna de uma especificação LOTOS na forma de uma sintaxe abstrata, age como um elemento facilitador para introdução desses novos componentes, dando ênfase à elaboração de um toolkit. Também o mapeamento da semântica dos operadores em classes C++ sugere uma alternativa na geração de um algoritmo de derivação de uma implementação automática a partir de uma especificação que pode ser investigada.

Anexo 1

Regras de derivação da Árvore de Execução

Este anexo contem as regras de derivação para gerar a Árvore de Execução a partir dos operadores LOTOS. Para cada operador *OP* é derivado um nó da Árvore. Note que se uma expressão EC contem uma sub-expressão EC_i , a Árvore resultante desta sub-expressão é referenciada no nó como AE_i. As regras são as seguintes :

- se *Op* é um operador ";" de prefixação com :
  EC= $g;EC_1$
  deriva-se a árvore

  $<Seq, g, AE_1>$

  EC= $i;EC_1$
  deriva-se a árvore
  $<IAct,AE1>$

- se *Op* é uma instanciação de processo (onde *Op* é o operador LOTOS implícito no nome do processo) com :
  EC = P $[g_1,...,g_n]$
  deriva-se a árvore

  $<Instance,P, <g_1,...,g_n>>$

- se *Op* é um operador de soma **choice ... in** com :
  EC= **choice** g **in** $[h_1,...,h_n]$ [] EC_1
  deriva-se a árvore

  $<GateSum,g, <h_1,...,h_n>,AE_1>$

- se *Op* é um operador de paralelismo **par ... in** com:

  EC= par g in $[h_1,...,h_n] \,||\, EC_1$

  deriva-se a árvore

  $<ParSyncronization,g, <h_1,...,h_n>,AE_1,...,AE_n>$

  onde $AE_1=AE_2=,...,=AE_n$

  EC= par g in $[h_1,...,h_n] \,||\,| EC_1$

  deriva-se a árvore

  $<ParInterleaving,g, <h_1,...,h_n>,AE_1,...,AE_n>$

  onde $AE_1=AE_2=,...,=AE_n$

  EC= par g in $[h_1,...,h_n] \,|[k_1,...,k_n]|\, EC_1$

  deriva-se a árvore

  $<ParComposition,g, <h_1,...,h_n>,<k_1,...,k_n>,AE_1,...,AE_n>$

  onde $AE_1=AE_2=,...,=AE_n$

- se *Op* é um operador **hide ... in** com :

  EC=**hide** $g_1,...,g_n$ **in** EC_1

  deriva-se a árvore

  $<Hide,g_1,...,g_n,AE_1>$

- se *Op* é um operador de habilitação ">>" com :

  $EC=EC_1>> EC_2$

  deriva-se a árvore

<Enable, AE_1, AE_2 >

- se *Op* é um operador de desabilitação "[>" com :

  $EC=EC_1$[> EC2

  deriva-se a árvore

<Disable, AE_1, AE_2 >

- se *Op* é um operador de escolha "[]" com :

  $EC=EC_1$ [] EC_2

  deriva-se a árvore

<Choice, AE_1,AE_2>

- se *Op* é um operador de composição paralela ||, |[]|, ||| com :

  $EC=EC_1||EC_2$

  deriva-se a árvore

<Sync,AE_1,AE_2>

$EC=EC_1|||EC_2$

deriva-se a árvore

<Interleaving,AE_1,AE_2>

$EC=EC_1|[g_1,...,g_n]|EC_2$

deriva-se a árvore

<*GeneralComposition*,<g_1,...,g_n>,AE_1,AE_2>

- se *Op* é um operador de terminação com insucesso, com :
  EC= **Stop**
  deriva-se a árvore

  <*Stop*>

- se *Op* é um operador de terminação com sucesso, com :
  EC= **Exit**
  deriva-se a árvore

  <*Exit*>

Anexo 2

Definição da função *Check*

Este anexo contém a definição da funcão *Check* para os diversos nós da ASA . Esta função define o algoritmo para se efetuar a análise semântica de um texto LOTOS a partir da ASA (Árvore Sintática Abstrata) equivalente. A função *Check* também define o algoritmo para geração da Árvore de Execução que é obtida à medida em que a ánalise é feita.

Além das funções Min e Max definidas no capítulo 4, usaremos também as seguintes funções auxiliares para manipulação de símbolos :

- *DeclarGate* : esta função tem como argumento uma lista de gates $<g_1,...,g_n>$ e um escopo *scp* e retorna uma lista de gates estendidos $<ge_1,...,ge_n>$. Sua definição é a seguinte :

  - Se $g_i = g_j$ para algum $j \neq i$ então erro, senão insere <gi,scp> $(1 \leq i \leq n)$ no conjunto NG do contexto estático.

- *DeclarProcess* : esta função tem como argumento uma lista de gates estendidos GE (que representam os parâmetros formais) , o identificador do processo P, a árvore de execução do processo AE, a funcionalidade do processo Func e o escopo SCP. Sua definição é a seguinte:

  - Se <P,SCP> in NP então erro (verifica se ja está definido).

  - Senão insere <P,GE,AE,Func,SCP> no conjunto NP do contexto estático .

Utilizando as funções auxiliares descritas, a função Check é descrita da seguinte maneira:

- Check(*SpecDef*) (SCP)
  DeclarProcess(A-Ident,A-IdentList,A-Exit,SCP,AE)
  DeclarGate(A-IdentList)(SCP')
  AE=Check(A-DefinitionBlock)(SCP)
  funcionalidade(AE)=funcionalidade(A-DefinitionBlock)
  Notas :

  - (a) A funcionalidade definida para a especificação deve ser igual à funcionalidade da AE, ou seja, funcionalidade(AE)=funcionalidade(A-Exit)

  - (b) SCP' é o escopo relativo à expressão de comportamento do nó A-DefinitionBlock, ou seja, SCP'=escope(AE) onde AE é a expressão de comportamento do bloco.

  - (c) As funções auxiliares DeclarProcess e DeclarGate alteram o contexto estático inserindo os identificadores do nome da especificação e dos gates declarados nos conjuntos NP e NG respectivamente.

- Check(*DefBlock*)(SCP)
  Check(A-LocalDef) (SCP)
  AE=Check(A-BehaviourExpress)(SCP')

funcionalidade(DefBlock)= funcionalidade(A-BehaviourExpress)

Notas :

- (a) A Árvore de Execução resultante da análise deste nó é a AE resultante da análise da Expressão de comportamento do bloco. O mesmo se aplica à funcionalidade.

- (b) O escopo SCP' é o escopo da expressão de comportamento do bloco. a funcionalidade do nó é a funcionalidade desta mesma expressão.

- Check(*LocalProcDef*) (SCP)

  Check(A-ProcessDef)(SCP)

  Check(A-LocalDef)(SCP)

- Check(*NullLocalDef*)(SCP)

  Nenhuma ação é tomada.

- Check(*ProcDef*)

  DeclarProcess(A-Ident,A-IdentList,NullList,A-Exit,SCP,AE)

  GateDeclar(A-IdentList)(SCP')

  AE=Check(A-DefinitionBlock)(SCP)

  Notas :

  - A funcionalidade definida para o processo deve ser igual à funcionalidade da AE, ou seja, funcionalidade(AE)=funcionalidade(A-Exit). A funcionalidade do nó A-Exit é EXITFUNC se o nó tem rótulo *Exit*, senão é NOEXITFUNC.

  - SCP' é o escopo relativo à expressão de comportamento do nó A-DefinitionBlock, ou seja, SCP'=escope(EC) onde EC é a expressão de comportamento do bloco.

- As funções auxiliares ProcessDeclar e DeclarGatesDeclar alteram o contexto estático inserindo os identificadores do nome do processo e dos gates declarados nos conjuntos NP e NG respectivamente.

- Check(Sync)(SCP)

  AE=<Sync,AE_1,AE_2>

  AE_1=Check(A-BehaviourExpress)

  AE_2=Check(A-BehaviourExpress)

  funcionalidade(Sync)=Min(funcionalidade(AE_1),funcionalidade (AE_2))

  Notas:

  - AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *Sync* , correspondentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 do nó *Sync*.

  - A funcionalidade do nó depende da funcionalidade das expressões AE_1 e AE_2. A função Min deve ser diferente de INDEFINIDA ou então teremos um erro semântico.

- Check(Interleaving)(SCP)

  AE=<*Interleaaving*,AE_1,AE_2>

  AE_1=Check(A-BehaviourExpress)

  AE_2=Check(A-BehaviourExpress)

  funcionalidade(*Interleaving*)=Min(funcionalidade(AE_1),funcionalidade (AE_2))

  Notas:

  - (a) AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *Interleaving*, correspon-

dentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 do nó.

- (b) A funcionalidade do nó depende da funcionalidade das expressões AE_1 e AE_2. A função Min deve ser diferente de INDEFINIDA ou então teremos um erro semântico.

- Check(*GeneralComposition*)(SCP)
  GL=CheckGates(A-IdentList)(SCP)
  AE=<*GeneralComposition*,GL,AE_1,AE_2>
  AE_1=Check(A-BehaviourExpress)
  AE_2=Check(A-BehaviourExpress)
  funcionalidade(*GeneralComposition*)=Min(funcionalidade(AE_1, funcionalidade(AE_2))
  Notas:

  - AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *GeneralComposition* , correspondentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 do nó.

  - A funcionalidade do nó depende da funcionalidade das expressões AE_1 e AE_2. A função Min deve ser diferente de INDEFINIDA ou então teremos um erro semântico.

- Check(*Choice*)(SCP)
  AE=<*Choice*,AE_1,AE_2>
  AE_1=Check(A-BehaviourExpress)(SCP)
  AE_2=Check(A-BehaviourExpress)(SCP)
  funcionalidade(*Choice*)=Max(funcionalidade(AE_1,funcionalidade(AE_2))
  Notas:

- AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *Choice* , correspondentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 do nó.

- A funcionalidade do nó depende da funcionalidade das expressões AE_1 e AE_2. A função Max deve diferente de INDEFINIDA ou então teremos um erro semântico.

• Check(*HidExpress*)(SCP)

GL=GateDeclar(A-IdentList,SCP')

AE=<*Hide*,GL,AE_1>

AE_1=Check(A-BehaviourExpress)(SCP)

funcionalidade(*HidExpress*) = funcionalidade(AE_1)

Notas:

- (a) A funcionalidade do nó depende da funcionalidade de AE_1.

- (b) GL é uma lista de identificadores de gate estendidos declarados para o escopo SCP',na forma <g,SCP'> onde SCP'=escopo(AE_1).

• Check(*Seq*)(SCP)

g=CheckGate(A-Ident)(SCP)

AE=<*Seq*,g,AE_1>

AE_1=Check(A-BehaviourExpress)(SCP)

funcionalidade(Seq)=funcionalidade(AE_1)

Notas:

- (a) A função CheckGate retorna um identificador estendido de gate $g \in NG$ com escopo(g)=SCP do contrário temos um erro semântico.

- Check(*IAct*) (SCP) }

  AE=<IAct,AE_1>

  AE_1=Check(A-BehaviourExpress)(SCP)

  funcionalidade(IAct)=funcionalidade(AE_1)

- Check(Stop)(SCP)

  AE=<*Stop*>

  funionalidade(*Stop*) = NOEXITFUNC

  Notas:

  - O nó Stop é um nó terminal (folha da árvore) no sentido em que não há sub-árvores referenciadas. Como tal, é também um nó terminal para a função *Check* , ou seja, não há chamada recursiva da função. Este nó define a funcionalidade da sub-expressão.

- Check(*EnableExpress*)(SCP)

  AE=<*EnableExpress*,AE_1,AE_2>

  AE_1= Check(A-BehaviourExpress)

  AE_2= Check(A-BehaviourExpress)

  funcionalidade(EnableExpress)= funcionalidade(AE_2)

  Notas:

  - AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *EnableExpress* , correspondentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 deste nó.

- Check(*DisableExpress*)(SCP)

  AE=<*DisableExpress*,AE_1,AE_2>

  AE_1= Check(A-BehaviourExpress)

  AE_2= Check(A-BehaviourExpress)

  funcionalidade(DisableExpress)=Max(funcionalidade(AE_1),

  funcionalidade(AE_2))

  Notas:

  - (a) AE_1 e AE_2 representam as duas expressões de comportamento compostas pelo operador *DisableExpress* , correspondentes respectivamente aos valores das variáveis Behaviour1 e Behaviour2 deste nó.

  - (b) A funcionalidade do nó depende da funcionalidade das expressões AE_1 e AE_2. A função Max deve ser diferente de INDEFINIDA ou então teremos um erro semântico.

- Check(*GateSum*)(SCP)

  GateDeclar(A-GateDeclar,SCP')

  AE=<*GateSum*,g_{11},<h_{11},...,h_{1n}>,AE_1>

  AE_1=Check(A-behaviourExpress)

Anexo 3

Definição das funções do núcleo do simulador

Neste anexo descrevemos as funções *Option* e *Step*. Estas funções compõem o núcleo do simulador LOTOS. Ambas as funçõoes são definidas a partir dos nós da Árvore de Execução (ver Anexo 1). Para cada nó da árvore é definido o algoritmo da função. Para efeito de comparação daremos além do nó a expressão de comportamento equivalente . A definição da função Options é a seguinte:

1. se $EC=g;EC_1$, $AE=<Seq,g,AE_1>$,

   então

   $Options(AE)= \{g\}$

2. se $EC=EC_1 [] EC_2$, $AE=<Choice,AE_1,AE_2>$,

   então

   $Options(AE)= Options(AE_1) \cup Options(AE_2)$

3. se EC= **par** *g* **in** $[h_1,...,h_n] \parallel EC_1$,

   $AE=<ParSyncronization, g,<h_1,...,h_n>,AE_1,...,AE_n>$

   então

   $Options(AE)=Options(AE_1)[h_1/g] \cap ... \cap Options(AE_n)[h_n/g]$

Nota : O termo $[h_i/g]$ significa substituir toda ocorrência de *g* por h_i dentro do conjunto obtido. O símbolo $\cap$ siginifica interseção entre conjuntos como normalmente.

4. se EC= **par** g **in** $[h_1,...,h_n]$ $\|[k_1,...,k_m]\|$ EC_1 ,

AE=<ParComposition, g,<$h_1,...,h_n$>, <$k_1,...,k_m$> ,$AE_1,...,AE_n$>

então

$K= \{k_i | \ i \leq 1 \leq m\} \cup \{ \delta\}$,

O_i=Options(AE_i)[h_i/g],

Options(AE)= ($O_1 \cup ... \cup O_n$ - K)$\cup$ ($O_1 \cap ... \cap O_n \cap$ K)

Nota : o conjunto O_i - K representa os eventos independentes na composição paralela. Os eventos nos quais as expressões compostas devem
se sincronizar é representado pelo conjunto $O_i \cap K$.

5. se EC= **par** g **in** $[h_1,...,h_n]$ $|||$ C_1 ,

AE=<*ParInterleaving*, g,<$h_1,...,h_n$>,$AE_1,...,AE_n$>

então

procede-se como no ítem anterior para K={δ}

6. se EC=$EC_1 \|_{EC2}$, AE=<*Sync*,AE_1,AE_2>

então

Options(AE)=Options(AE_1) $\cap$ Options(AE_2)

7. se EC=$EC_1 ||| EC_2$, AE=<*Interleaving*,AE_1,AE_2>

então

Options(AE)=(Options(AE_1) $\cup$ Options(AE_2) - { δ}) $\cup$
(Options(AE_1) $\cap$ Options(AE_2) $\cap$ { δ})

8. se EC=$EC_1 \|[g_1,...,g_n]\| EC_2$,

AE=<*GeneralComposition*,<$k_1,...,k_n$>, AE_1,AE_2 >

então

$K=\{k_i | i \leq 1 \leq m\} \cup \{\delta\}$,

O_i=Options(AE_i),

Options(AE)= ($O_1 \cup O_2$ - K)$\cup$ ($O_1 \cap O_2 \cap$ K)

9. se EC= **choice** *g* **in** $[h_1,\ldots,h_n]$ [] EC_1 ,AE=<*Gate-Sum*,$gh_1,\ldots,h_n$],EC_1 >

então

Options(AE)=Options(AE_1) $[h_1/g] \cup \ldots \cup$ Options(AE_n)$[h_n/g]$

10. se EC = P$[g_1,\ldots,g_n]$, AE=<*Instance*,P,<$g_1,\ldots,g_n$>>

então

Options(AE)=Options(AE_1)$[g_i/f_i]$

Nota : AE_1 é Árvore de execução referenciada no identificador de processo estendido P. O termo $[g_i/f_i]$ representa a substituição de toda ocorrência em AE de f_i por g_i onde f_i é o iésimo parâmetro de gate formal de P e g_i é o gate real utilizado na instanciação do processo que ocupa a mesma posição.

11. se EC=Stop, AE=<*Stop*>

então

Options(AE)={}

12. se EC=Exit, AE=<*Exit*>

então

Options(AE)={δ }

Nota : o símbolo δ representa o evento interno relativo à terminação com
sucesso.

13. se EC= **hide** $g_1,...,g_n$ **in** EC_1, AE=<*Hide*,$g_1,\ldots,g_n,EC_1$>
então
Options(AE)=Options(AE_1) [i/g_j]

Nota : o termo [i/g_j] significa substituir toda ocorrência de g_j ($1 \leq j \leq n$)
pelo gate interno *i*.

14. se EC=EC_1>> EC_2, AE=<*Enable*,AE_1,AE_2>
então
Options(AE)=Options(AE_1)

15. se EC=EC_1 [> EC_2, AE=<Disable,AE_1,AE_2>
então
Options(AE)=Options(AE_1) ∪ Options(AE_2)

16. se EC=i;EC_1, AE=<IAct,AE_1>
então
Options(AE)= i

A definição da função *Next* será feita da mesma maneira que para a função *Options* com a exceção de que não repetiremos aqui a expressão de comportamento original mas apenas o nó da AE. A definição é a seguinte:

1. AE=<*Seq*,g,AE_1>

se a=evento(g) então

Next (a,AE) = <True,AE_1>

senão

Next(a,AE) = <False,AE>

Nota : evento(g) significa o evento ofertado no gate g .

2. AE= <*Instance*,P,<g_1,...,gn>>

se Next (a,AE_1 [g_i/f_i]) = <True,AE_1'>

então

Next (a,AE) =<True,AE_1'>

senão

Next (a,AE) =<False,AE>

Nota: AE_1 é a Árvore de Execução que representa o processo.

3. AE= <*GateSum*,g,<h_1,...,h_n>,EC_1>

se Next (a,AE_1 [g_i/g]) = True,AE_1' para algum i, $1 \leq i \leq n$

então

Next (a,AE) = <True,AE_1'>

senão

Next (a,AE) = <False,AE>

4. AE= <*ParSyncronization*,g,<h_1,...,h_n> ,AE_1 ,..,AE_n >

se Next (a,AE_i [h_i/g]) =< True,AE_i'> para todo i, $1 \leq i \leq n$

então

Next (a,AE) =<True,AE'> , com

AE'= < *ParSyncronization*,g<,h_1,..,h_n> ,AE_1',..,AE_n'>

senão

Next (a,AE)= <False,AE>

5. AE= <*ParInterleaving*,g,<h_1,...,h_n> ,AE_1 ,..,AE_n>

se Next (a,AE_i[h_i/g])=<True,AE_i'> para algum i, $1 \leq i \leq n$

então

Next (a,AE)=<True,AE'>, com

AE'= < *ParInterleaving*,g,<h_1,...,h_n> ,AE_1',..,AE_n'> , e

AE_i'= AE_i '' se Next (a,AE_i[h_i/g])=<True,AE_i">

senão AE_i'=AE_i

senão

Next (a,AE)= <False,AE>

6. AE= < *ParComposition*,g,<h_1,...,h_n>,<k_1,...,k_m >,AE_1 ,...,AE_n>

se a $\in$ { k_1,...,k_m} $\cup$ {δ} procede-se como em (4).

se a $\notin$ { k_1,...,k_m } $\cup$ {δ} procede-se como em (5).

7. AE= <*Hide*,<g_1,...,g_n>,AE_1 >

se Next(a,AE_1)=True,AE_1'

então

Next (a,AE)=True,AE', com

AE'= <*Hide*,<g_1,...,g_n>,AE_1'>

senão

Next (a,AE)=<False,AE>

8. AE= <*Enable*,AE_1 ,AE_2>

se a=evento(δ) e Next (a,AE_1)=True

então Next (a,AE)=< True,AE_2 >

senão

se Next (a,AE_1)=<True,AE_1' >

Next(a,AE)=<True,AE'>, com

AE'= <*Enable*,AE_1',AE_2 >

senão

Next (a,AE)=<False,AE>

9. AE=<*Disable*,AE_1 ,AE_2 >

se a=evento(δ) e Next (a,AE_1)=<True,<*Stop*>>

então Next (a,AE)=<True, <*Stop*>>

senão

se Next (a,AE_1)=<True,AE_1' >

então

Next (a,AE)=<True,AE' >, com

AE' = <*Disable*,AE_1',AE_2 >

senão

se Next (a,AE_2)=True,AE_2'

então

Next(a,AE)=<True,AE_2'>, com

senão

Next (a,AE)=<False,AE>

10. AE= <*Choice*,AE_1 ,AE_2 >

se Next (a,AE_1)=<True,AE_1' >

então

Next (a,AE) = <True,AE_1' >

senão

se Next (a,AE_2)=<True,AE_2' >

então

Next (a,AE)=<True,AE_2' >

senão

Next (a,AE) =<False,AE>

11. AE=<*Sync*,AE_1 ,AE_2 >

se Next (a,AE_1)=<True,AE_1'>, e

Next (a,AE_2)=<True,AE_2'>

então

Next (a,AE)=<True,AE'>, com

AE'=<*Sync*,AE_1',AE_2'>

senão

Next (a,AE)=<False,AE>

12. AE=<*Interleaving*,AE_1 ,AE_2 >

se a=evento(δ), procede como (11).

senão

se Next (a,AE_1)=<True,AE_1' >

então

Next (a,AE)=<True,AE'>, com

AE'= <*Interleaving*,AE_1',AE_2 >

senão

se Next (a,AE_2)=<True,AE_2' >

então

Next (a,AE)=<True,AE'>, com

AE'= <*Iterleaving*,AE_1 ,AE_2'>

senão

Next (a,AE) =<False,AE>

13. AE=<*GeneralComposition*,<g_1,...,g_n>, AE_1 ,AE_2 >

se a $\in$ {g_1,...,g_n} procede como em (11)

senão procede como em (12)

14. AE= <*Stop*>

Next(a,AE) = <False,AE>

Nota : O nó Stop não admite nenhuma evolução

15 . AE= <*Exit*>

se a=evento{δ}

então

Next (a,AE)=<True,<*Stop*>>

senão

Next (a,AE)= <False,AE>

Anexo 4

Definição da Árvore Sintática Abstrata

A-specification :==	*SpecDef (1)*	
	(SpecId :	A-Ident
	FormalGates :	A-IdentList
	FormalValues :	A-IdentDeclarations
	Functionality :	A-Exit
	GlobalType :	A-GlobalTypeDef
	Block :	A-DefinitionBlock
	)	
A-GlobalTypeDef :==	*GlobalDef (2)*	
	(GlobalData :	A-DataTypeDef
	ProxGlobalDef :	A-GlobalTypeDef
	)	
	|	
	NullDef (3)	
	()	
A-definitionBlock :==	*DefBlock (4)*	
	(Behaviour :	A-BehaviourExpress
	LocalDefinitions :	A-LocalDef
	)	
A-LocalDef :==	*LocalProcDef (5)*	
	(ProcessDefinition :	A-ProcessDef

ProxLocalDef : A-LocalDef

)

|

LocalDataDef (6)

(LocalData : A-DataTypeDef

ProxLocalDef : A-LocalDef

)

|

NullLocalDef (3)

( )

A-DataTypeDef :== *LibDef (7)*

(TypeList : A-IdentList

)

|

PSpec (8)

(TypeName : A-Ident

TypeUnion : A-IdentList

FormalSorts : A-IdentList

FormalOperations : A-Operations

FormalEquations A-Equations

Sorts : A-IdentList

Operations : A-Operations

Equations : A-Equations

)

|

Actualization (9)

(TypeName : A-Ident *BaseType:*
A-Ident

TypeUnion: A-IdentList

SortReplace: A-SortPairList

OperationReplace A-OperationPairList

)

|

Rename (10)

(TypeName : A-Ident

BaseType : A-Ident

SortRename A-SortPairList

OperationRename A-OperationPairList

)

A-SortPairList :== *SortPairList (11)*

FromSort : A-Ident

ToSort : A-Ident

NextPair : A-SortPairList

)

|

NullSortPair (3)

( )

A-OperationPairList :== *OperationPairList (12)*

(FromOperation : A-Ident

ToOperation : A-Ident

NextPair : A-OperationPairList

)

|

NullOperationPair (3)

( )

A-Operations :== *OpList (13)*

(OperationPosition : A-OpDescriptors

ArgumentSorts : A-ArgSortList

Result : A-ResultSort

NextOperatios : A-Operations

)

|

NullOpList (3)

( )

A-OpDescriptors :== *OperInfix (14)*

(OperationName : A-Ident

NextOperPosition : A-OpDescriptors

)

|

OperNulary (15)

(OperationName : A-Ident

NextOperPosition : A-OpDescriptors

)

|

OperPostfix (16)

(OperationName : A-Ident

NextOperPosition : A-OpDescriptors

)

|

NullOpDesc (3)

()

A-Equations :== *Equations (18)*

(GenericVariableDeclar : A-IdentDeclarations

EquationsDesc : A-EquationLists

A-EquationLists :== *EqLists (19)*

	(ParticularVarDeclar :	A-IdentDeclarations
	EquationsSort :	A-Ident
	EquationIdentities:	A-EqList
	NextEquations :	A-EquationLists
	)	
	|	
	NullEqList (3)	
	()	
A-EqList :==	*EqList (20)*	
	(Premisses :	A-Premisses
	RightSide :	A-ValueExpress
	LeftSide :	A-ValueExpress
	NextEquation :	A-EqList
	(	
	|	
	NullEqList (3)	
	( )	
A-Premisse :=	*SimpleEquation (21)*	
	(RightSide :	A-ValueExpress
	Leftside :	A-ValueExpress
	NextPremiss	A-Premmisses

)

|

BoolExpress (22)

(BooleanExpression : A-ValueExpress

NextPremiss: A-Premmisses

)

|

NullPremmiss (3)

( )

A-ProcessDef :== *ProcDef (23)*

(ProcId : A-Ident

FormalGates : A-IdentList

FormalValues : A-IdentDeclarations

Functionality : A-Exit

Block : A-DefinitionBlock

)

A-Exit :== *NoExit (24)*

( )

|

Exit (25)

( )

|

ExitList (26)

(Sorts : A-IdentList

)

A-BehaviourExpress :== **A-LocalDefExpress**

|

A-GateSumExpress

|

A-ValueSumExpress

|

A-ParExpress

|

A-ParCompExpress

|

A-HidingExpress

|

A-DisableExpress

|

A-PrefixExpress

|

A-AtomicExpress

|

A-ChoiceExpress

A-LocalDefExpress :== *LocalAssignEpress (27)*

(Assignments : A-IdentEqns

Behaviour : A-BehaviourExpress

)

A-GateSumExpress :== *GateSum (28)*

(SumDomain : A-GateDeclars

behaviour : A-BehaviourExpress

)

A-ValueSumExpress :== *ValueSum (29)*

(SumDomain : A-IdentDeclarations

Behaviour : A-BehaviourExpress

)

A-ParExpress :== *ParComposition (30)*

(ParDomain : A-GateDeclar

SyncGates : A-IdentList

Behaviour: A-BeheviourExpress

)

|

ParInterleaving (301)

	(ParDomain :	A-GateDeclar
	Behaviour :	A-BehaviourExpress
	)	
	|	
	ParSyncronization (302)	
	(ParDomain :	A-GateDeclar
	Behaviour :	A-BehaviourExpress
	)	
A-HidingExpress :==	*HidExpress(31)*	
	(InternalGates :	A-GateList
	Behaviour :	A-BehaviourExpress
	)	
A-EnableExpress :==	*AcceptExpress (49)*	
	ValueIdents :	A-IdentDeclars
	Behaviour :	A-BehaviourExpress
	EnabledBehaviour :	A-BehaviourExpress
	)	
	|	
	EnableExpress (32)	
	(Behaviour :	A-BehaviourExpress
	EnabledBehaviour :	A-BehaviourExpress

)

A-DisableExpress :== *DisableExpress (132)*

(Behaviour : A-BehaviourExpress

DisableBehaviour : A-BehaviourExpress

)

A-ParCompExpress :== *Sync (33)*

(Behaviour1 : A-BehaviourExpress

Behaviour2 : A-BehaviourExpress

)

|

Interleaving (34)

(Behaviour1 : A-BehaviourExpress

Behaviour2 : A-BehaviourExpress

)

|

GeneralComposition (35)

(SyncGates : A-IdentList

Behaviour1 : A-BehaviourExpress

Behaviour2 : A-BehaviourExpress

)

|

A-ChoiceExpress :== *Choice (48)*

(Behaviour1 : A-BehaviourExpress

Behaviour2 : A-BehaviourExpress

)

A-GuardedExpress :== *Guard (36)*

(Guard : A-Premisses

Behaviour : A-BehaviourExpress

)

A-PrefixExpress :== *Seq (37)*

(GateName : A-Ident

Offers : A-ExperimentOffer

Predicate : A-Premiss

Behaviour : A-BehaviourExpress

)

|

InternalAct (377)

( )

A-AtomicExpress :== *Stop (38)*

( ) :

|

Exit (39)

( )

|

ExitValue (40)

(ValueList : A-ExitValueList

)

|

Instanciation (41)

(ProcessId : A-Ident

ActualGates : A-IdentList

ActualValues : A-ValueExpressList

)

A-ExitValueList :== *ExitParValue (411)*

(Value : A-ValueExpress

NextValue : A-ExitValueList

)

|

ExitParAny (412)

(SortName : A-Ident

NextValue : A-ExitValueList

)

|

NullExitValList

( )

A-IdentList :== *IdentList (42)*

(IdentName : A-Ident

NextIdent : A-IdentList

)

A-ValueExpress :== *ValueId (43)*

(VarName : A-Ident

Sort: A-Ident

)

|

NullaryOper (44)

(OperName : A-Ident

Sort : A-Ident

)

|

InfixOper (45)

(OperName : A-Ident

Sort : A-Ident

LeftValue : A-ValueExpress

RightValue : A-ValueExpress

)

|

PostFixOper (46)

(OperName : A-Ident

Sort : A-Ident

ValueList: A-ValueExpList

)

A-ValueExpList :== *VExpList (47)*

(Value : A-ValueExpress

NextValue : A-ValueExpList

)

|

NullVexpList (3)

( ) :

A-Identdeclarations :== *IdentDeclar (50)*

(VarNames : A-IdentList

Sort : A-Ident

NextDeclar : A-IdentDeclaration

)

A-GateDeclar :== *GateDeclar*

GateTuple : A-IdentList

NextDeclar : A-GateDeclar

)

|

NullGateDeclar

() :

A-IdentEqns :== *IdentEqns (51)*

(Variables : A-IdentDeclarations

Values : A-ValueExpress

NextIdentEqn : A-IdentEqns

)

|

NullIdentEqn (3)

()

Anexo 5

Manual de utilização das ferramentas

Este Anexo descreve o procedimento de instalação e utilização das ferramentas que compoem o Ambiente de Desenvolviento Lotos (ADL) . Para se utilizar destas ferramentas é necessário o seguinte:

- Computador pessoal tipo IBM PC ou compatível com 640 kb de memória.
- DOS (Disk Operating System) versão 3.1 ou superior
- Disco Rígido com espaço suficiente para 500kb (tamanho máximo dos arquivos que formam o ADL) no caso de utilização do modo "integrado".

5.1 Descrição dos arquivos

O ADL é composto de dos seguintes arquivos:

- LC.EXE : arquivo executável do compilador LOTOS
- SIMU.EXE : arquivo executável do simulador LOTOS.
- LCRIS.EXE : arquivo executavel do editor de texto e front-end para uso integrado do ADL.
- EDITERR.MSG : arquivo contendo as mensagens de erro do editor.
- HELP.SCR : arquivo contendo as telas para a janela de HELP do editor.

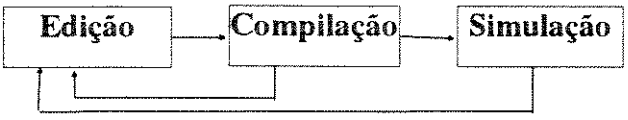
5.2 Operação do ADL

O ADL pode ser utilizado de duas maneiras :

- integrado: todas as funções são solicitadas a partir de teclas especiais dentro do editor. Neste modo o usuário não distingue os diversos arquivos que compoem o ADL.

- não integrado : O usuário enxerga as diversas ferramentas que compõem o ADL (o editor, o compilador e o simulador) e executa-os isoladamente.

No modo não integrado, cada ferramenta pode ser chamada isoladamente. O processo de utilização é o seguinte :



O ciclo edição-compilação-simulação pode ser repetido tantas vezes quanto necessário. Em cada chamada de qualquer das ferramentas devem ser observadas a sintaxe de chamada da ferramenta em questão conforme descrita adiante. No modo integrado, o usuário precisa conhecer apenas os comandos do editor, que incluem chamadas automáticas (escondidas) para as outras ferramentas.

5.3 Utilizando o compilador

A sintaxe de utilização do compilador é a seguinte :

- **LC** [-anome1.ext] [-lnome.ext] [**-d**] [**-s**] arquivo

onde,

- **-a** : flag indicando que deve ser gerado um arquivo com a ASA (Árvore Sintática Abstrata) a ser utilizada pelo simulador. Deve ser utilizada quando não se tem mais erros sintáticos/semânticos e o programa está pronto para simulação. Seguido do flag -a deve vir o nome do arquivo que conterá a ASA. Se não for fornecida uma extensão para o arquivo, será gerada a extensão .AST automaticamente.

- **-l** : flag indicando que deve ser gerado um arquivo listando o programa após processado pelo compilador. Este arquivo conterá eventuais erros de compilação. O nome do arquivo que conterá a lista é fornecido após o flag **l**.

- **-s** : gera uma lista de todos os símbolos declarados na especificação. Estes serão escritos no stdout (standard output) do DOS.

- **-d** : flag de depuração. Utilizado quando se deseja ver quais as regras de produção da gramática LOTOS que foram reconhecidas. Útil na depuração do próprio simulador para identificar reconhecimento errôneo de sentenças.

- arquivo.ext : nome do arquivo a ser compilado. Se não for especificada a extensão, é assumido a extensão default .l

Exemplos de utilização:

- LC -atrans trans
  É feita a compilação do arquivo "trans.l" (extensão default) e é gerado um arquivo "trans.ast" contendo a ASA equivalente.

- LC trans.lot
  É feita a compilação do arquivo "trans.lot" . Eventuais erros são escritos no stdout do DOS.

- LC -ltrans.lis trans
  É feita a compilação do arquivo "trans.l" e é gerado o arquivo "trans.lis" contendo uma listagem do programa com indicação de eventuais erros de compilação.

5.4 Utilizando o simulador

Para se utilizar o simulador é necessário que se tenha compilado previamente a especificação LOTOS a ser simulada. Para isto é necessário que se utilize o compilador com a opção **-a** (ver item anterior) de modo a se ter um arquivo contendo a ASA equivalente. A sintaxe do comando do simulador é :

- SIMU [nome[.ext]]

onde nome.ext é o nome do arquivo que contém a ASA equivalente da especificação a ser simulada. No caso de não ser especificada a extensão do arquivo (omitir o .ext no nome) é assumido a extensão padrão .ast . uma vez

iniciado o processo de simulação o usuário pode executar vários comandos durante a execução do simulador. Os comandos são os seguintes:

- **Help** : imprime uma tela de ajuda ao usuário, contendo uma relação dos comandos disponíveis.

- **start** n : inicia a simulação de um dado processo. Note que se uma especificação contém vários processos, a simulação pode se iniciar em qualquer um destes processos independente da hierarquia. O processo a ser simulado é identificado pelo número "n".

- **menu** : imprime um menu com as opções de evolução da especificação (eventos disponíveis) . As opções do menu são numeradas de modo a identificá-las através do respectivo número.

- **step** n : executa um passo da simulação. É necessário especificar qual qual o evento associado ao passo de execução. O evento "n" é identificado pelo respectivo número associado no menu de opções.

- **showstep** n : mostra passo será executado se um determinado evento "n" for escolhido.

- **Exit** : termina a simulação de um processo ou sai do simulador se nenhum processo estiver sendo simulado.

- **State** : mostra o estado atual da simulação. O estado é descrito como uma expressão de comportamento LOTOS.

5.5 Utilizando o Editor

O editor de texto é utilizado para se gerar o texto da especificação. Além disso, o editor possui comandos específicos para ativação do compilador e simulador de modo a se ter um ambiente integrada de de simulação LOTOS. A sintaxe do comando de edição é :

- Lcris [nomearq]

Não é assumido nenhum nome pré-definido para o nome do arquivo , sendo portanto um parâmetro que deve ser especificado pelo usuário. Os comandos do editor são classificados em comandos *diretos* e comandos *prefixados D. Os comandos diretos* são aqueles que são obtidos pressionando-se a tecla <crtl> (control) e uma letra especifica. Os comandos *prefixados* são obtidos pressionando-se simultaneamente a tecla <crtl> e uma das letras k,q,o seguido da letra do comando . Um exemplo de um comando prefixado é <crtl>qr (control q seguido de r) que vai para o início do texto.

Os comandos diretos são os seguintes:

- ^A : move o cursor uma palavra para a esquerda
- ^F : move o cursor uma palavra para a direita
- ^S : move o cursor uma letra para a esquerda
- ^D : move o cursor uma letra para a direita

- ^E : move o cursor para a linha superior
- ^X : move o cursor para a linha inferior
- ^R : move o cursor para a pagina anterior
- ^C : move o cursor para a próxima página
- ^W : move o texto uma linha para cima (scroll up)
- ^Z : move o texto uma linha para baixo (scroll down)
- ^G, DEL : elimina o caracter sob o cursor.
- BKSP : elimina o caracter à esquerda do cursor
- ^N : insere uma linha em branco
- ^Y : elimina a linha que contem o cursor.
- ^I : insere um tabulador
- ^T : elimina a palavra sob o cursor
- ^V : chaveia o modo de inserção
- ^L : procura próxima ocorrência.
- ^J : vai para o início/fim da linha
- ^U : cancela comando
- ^P : insere caracter de controle
- RETURN : insere uma linha nova
- ESC : desfaz última mudança

Comandos prefixados de Q

- ^QA : procura e substitui
- ^QF : procura ocorrência de uma palavra
- ^QB : vai para início do bloco
- ^QK : vai para o fim do bloco
- ^QS : vai para inicio da linha
- ^QD : vai para o começo da linha seguinte
- ^QR : vai para o início do arquivo
- ^QC : vai para o fim do arquivo
- ^QY : elimina a linha à direita
- ^Q I : entra em modo de identação
- ^Q n : vai para a marca n
- ^QJ : vai para a próxima marca

Comandos prefixados de K

- ^KD : salva/abre arquivo
- ^KS : salva arquivo
- ^KX : retorna ao DOS (sai do editor)
- ^KQ : abandona arquivo corrente, abre um novo arquivo

- ^K n : põe marca n
- ^KM : põe uma marca
- ^KB : delimita início de bloco
- ^KK : delimita fim de bloco
- ^KH : esconde bloco
- ^KC : copia bloco
- ^KV : move bloco
- ^KY : elimina bloco
- ^KW : escreve bloco em arquivo
- ^KR : le bloco de arquivo
- ^KT : define tamanho do tabulador

Comandos precedidos de ^O :

- ^OL : ajusta margem à esquerda
- ^OR : ajusta margem à esquerda
- ^OC : centraliza linha
- ^OK : muda maiúscula/minúscula
- ^OI : vai para a coluna n
- ^ON: vai para a linha n

Comandos com teclas especiais F1-F10 :

- F1 auxílio
- F1 auxílio numa palavra chave sob o cursor
- F2 salva arquivo
- F5 fecha a janela de erros
- F6 chaveia janela edição/erro
- F7 : ativa o analisador sintático/semântico
- F9 : ativa o compilador e simulador
- F10 : acessa o menu.

Bibliografia

[Aho 77] A. V. Aho, J. D. Ullman, "*Principles of Compiler Design*", Addison Wesley, Massachussets, 1977

[Axe 85] A.T.Schreiner, H. G. Friedman: "*Introduction to Compiler Construction with Unix*", Prentice Hall

[Bloom 88] R.E. Bloomfield et all. : " Safety Critical Software " , in Software Engineering Conference, London, June 1988.

[Bol 87] T. Bolognesi, S. Smolka : "*Fundamental Results For The Verification Of Observational Equivalence* " , VII IFIP, Zurich 1987

[Bol 87b] T. Bolognesi, E. Brinksma : "Introduction to the ISO Specification Language LOTOS ", in Computer Networks and ISDN Systems, Vol 14, 1987

[BFL 86] J.P. Briand, M.C. Fehri, L. Logrippo, A. Obaid: "*Structure and Use of a LOTOS Interpreter*", Université de Montreal

[Bri 86a] E. Brinksma, G. Scollo, C. Vissers: "*Experience With And Future of LOTOS as a Specification Language*", Memorandum INF 86-13, Twenty University of Technology, 1986

[Bri 86b] E. Brinksma, G. Scollo, C. Steenbergen: "*LOTOS Specifications, Their Implementation and Their Tests*", in Protocol Specification, Testing and Verification, 1986

[Boc 89] G.V. Bochman, Q. Gao, C. Wu: "*On the Distributed Implementation of LOTOS*", FORTE 89, Vancouver, 1989

[Bla 89] S. Black: "*Objects and LOTOS*", FORTE 89, Vancouver, 1989

[Bri 86] J.P. Briand, M.C. Fehri, L. Loggripo, L. Obaid: "*Executing LOTOS Specifications*", in Protocol Specification, Testing and Verification, North Holland, 1986

[Eij 89] P. V. Eijk :"*Software Tools for the Specification Language Lotos*", Phd. Thesis, Twenty University of Technology, 1989

[Eij 86] P. Van Eijk: "*A Comparison of Behavioural Language Simulators*", in Protocol Specification, Testing and Verification, 1986

[Eij 89b] P. Van Eijk: "*LOTOS Tools Based on the Cornel Synthesizer Generator*", Protocol Specification, Testing and Verification, 1989

[Fre 87]Peter Freeman : " *Software Perspectives: The System is the Message* ", Addison-Wesley Publishing Company, 1987

[Gar 89] H. Garavel: "*Compilation of LOTOS Abstract Data Types*", FORTE'89

[Gil] D. Gilbert: "Executable LOTOS: using PARLOG to implement a FDT", Imperial College of Science and Technolog, London

[Ghl 88] R. Guillemont, L. Logrippo, M. Haj-Hussein: "*Executing Large LOTOS Specifications*", in Protocol Specification, Testing and Verification VIII, North Holland, 1988

[Gll 88] R. Guillemont, L. Logrippo: "*Derivation of Useful Execution Trees From LOTOS Specifications by Using an Interpreter*", FORTE'88

[Gil 89] D. Gilbert: "*Implementing LOTOS in Parlog*",MSc Thesis, Imperial College, London

[Got 88] R.Gotzhein: " *Specifying Abstract Data Types with LOTOS*", internal paper, Université de Montreal

[Hal 90] A. Hall: "*Seven Myths of Formal Methods*", in IEEE Software, September 1990

[Hee 90] J. Heering et all : " Algebraic Specification ", ACM Press, New York, 1990

[Her 85] H. Ehrig, B. Mahr: "*Fundamentals of Algebraic Specifications*", Springer-Verlag, Berlin, 1985

[Hoa 85] C. A. Hoare: "*Communicating Sequential Processes*", Prentice Hall, UK 1985

[Hen 80] K. Heniger: "*Specifying Software Requirements for Complex Systems: new techniques and their applications*", in IEEE Transaction on Software Engineering, janeiro, 1980

[ISO 87] ISO 8807 - Information Processing Systems - Open Systems Interconnection - LOTOS - a formal technique based on the temporal ordering of observational behaviours

[Joh 78] S.C. Johnson: "*YACC: Yet Another Compiler-Compiler*", in Unix Programmers Manual, Bell Labs, 1978

[Kbe 70] D. E. Knuth, P. B. Bendix: "*Simple Word Problems in Universal Algebras*", in Computational Problems in Abstract Algebra, Pergamon Press, 1970

[Kor 89] Korean Contribution to ISO: "*An Object-Oriented Extension of LOTOS for Distributed Processing*" ISO/IEC JTC1 Sc21 WG7

[Led 87] G. J. Leduc, "*The intertwining of Data Type and Processes in LOTOS*", Protocol Specification, Testing and Verification, 1987

[Les 78] M. E. Lesk, E. Schmidt: "*Lex: a Lexical analyser generator*", in Unix Programmers Manual, Bell Labs, 1978

[Log 85] L. Logrippo, D. Simon, H. Ural: "*Executable Description of The OSI Transport Service in Prolog*", in Protocol Specification, Testing and Verification , North-Holland, 1985

[Log 88] L. Logrippo et al: "*An interpreter for LOTOS: a specification language for distributed systems*", FORTE'88

[LSc 88] J. Van der Langemaat, G. Scollo: "*On the use of LOTOS for the formal description of a transport protocol*", FORTE'88

[Lus 89] E. Lusak, S. Rudkin, C. Smith: "*An Object-Oriented Interpretation of LOTOS*", FORTE 89, Vancouver, 1989

[May 88] T. Mayr: "*Specification of Object-Oriented Systems in LOTOS*", FORTE 88

[Man 89] J. A. Manas, T. de Miguel, H. Van Thicken: "*The implementation of a specification language for OSI systems*", in Formal Description Technique LOTOS, North Holland, 1989

[Mdm 88] J. A. Manas, T. de Miguel: "*From LOTOS to C*", FORTE'88

[Men 88] S. Mendes, T. Aguiar:"*Métodos para Especificação de Sistemas* " , in EBAI, 1988

[Mil 79] R. Milner:"*Calculus of Communicating Systems*" Springer-Verlag, 1979

[Mil 86] R. Milner:"*Process Constructors and Interpretations*", in IFIP vol 10, North Holland, 1986

[Pap] G. Pappalardo: "*Experiences with verification and simulation tools for behavioural languages*"

[Ped 88] J. S. Perderson, M. H. Klein: "*Using the Vienna Development Method to formalize a communication protocol*", Carnegie Mellow University Technical Report

[Que 88] J. Quemada, S. Pavon, A. Fernandes: "*Transforming LOTOS specifications with LOLA - The parameterized expansion*", in FORTE'88

[Sin 80] M. Van Sinderen: "*The application of LOTOS for the formal description of the ISO session layer*", FORTE'89

[Sid 83] D. Sidhu: "*Protocol Verification via Executable Logic Specifications*", in Protocol Specification, Testing and Verification, North Holland, 1983

[Som 89] I. Sommerville: *"Software Engineering", Addison Wesley, 1989*

[Stroup] B. Stroustrup: "The C++ programming language", Addison Wesley, 1990

[Tre 89] J. Tretmans: *"HIPPO: a LOTOS simulator"*, in Formal Description Technique LOTOS, North Holland, 1989

[Turb] Turbo Pascal Toolbox - Borland - 1989

[Vij 90] V. Varadharajan: *"Use of a Formal Description Technique in the specification of authentication procotol"*, in Computer Standard and Interfaces, 1989/90