MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO DE PESQUISAS ESPACIAIS

AUTORIZAÇÃO PARA PUBLICAÇÃO
AUTHORIZATION FOR PUBLICATION

PALAVRAS CHAVES/KEY WORDS

AUTORES/AUTHORS

MODELO OPERACIONAL / MÉTODO JSD
ESPECIFICAÇÃO EXECUTÁVEL

AUTORIZADA POR/AUTHORIZED BY

Ralf Gielow
Pres. Cons. P.G.

AUTOR RESPONSAVEL
RESPONSIBLE AUTHOR

Ana Maria Ambrosio

DISTRIBUIÇÃO/DISTRIBUTION

- ☐ INTERNA / INTERNAL
- ☒ EXTERNA / EXTERNAL
- ☐ RESTRITA / RESTRICTED

REVISADA POR / REVISED BY

Flávio Roberto D. Velasco

CDU/UDC

519.87

DATA / DATE

Setembro 1989

PUBLICAÇÃO Nº
PUBLICATION NO

INPE-4915-TDL/381

ORIGEM
ORIGIN

PG/DSS

PROJETO
PROJECT

FRH/CAP

Nº DE PAG.
NO OF PAGES

207

ULTIMA PAG.
LAST PAGE

C9

VERSÃO
VERSION

Nº DE MAPAS
NO OF MAPS

TÍTULO/TITLE

UM SISTEMA PARA EXECUÇÃO
DE ESPECIFICAÇÕES JSD

AUTORES/AUTHORSHIP

Ana Maria Ambrosio

RESUMO - NOTAS / ABSTRACT - NOTES

Modelos operacionais para desenvolvimento de softaware foram propostos como uma alternativa aos modelos convencionais (especifica, projeta e implementa). Um dos princípios básicos dos modelos operacionais é que é possível executar diretamente a especificação de requisitos de software. Neste trabalho, o método JSD ("Jackson System Development") é considerado como um modelo operacional e, para tanto, uma linguagem e um esquema são propostos que possibilitem a execução de especificações JSD. É, também, implementado um sistema que analisa especificações e permite que elas sejam executadas.

OBSERVAÇÕES / REMARKS

Dissertação de Mestrado em Computação Aplicada, aprovada em 8 de junho de 1988.

Aprovada pela Banca Examinadora
em cumprimento a requisito exigido
para a obtenção do Título de Mestre
em Computação Aplicada

Dr. Luiz Alberto Vieira Dias

Presidente

Dr. Flávio Roberto Dias Velasco

Orientador

Dr. Mário Jino

Membro da Banca
-convidado-

Dr. Paulo Ouvera Simoni

Membro da Banca

Candidato: Ana Maria Ambrosio

São José dos Campos, 08 de junho de 1988

Dedico este trabalho à Candida, minha mãe e à Ni, minha irmã.

AGRADECIMENTOS

Agradeço a Dr. Flávio Roberto Dias Velasco pela orientação, incentivo e interesse. Meus agradecimentos também a Dr. Eduardo W. Bergamini pela oportunidade de realizar este trabalho no departamento; à Ana Maria Vadô e Maria Aparecida Monteiro pelo auxílio na formatação deste documento, ao Sérgio Rosim e Naoto Shitara pela cordialidade em emprestarem seu microcomputador durante a fase de desenvolvimento; ao Mário, ao Miguel, ao Leonardo e a todos os meus amigos que de alguma forma colaboraram para a conclusão deste trabalho.

ABSTRACT

The operational approach for software development was suggested as an alternative to the conventional approach (specify, project and implement). One of the basic principles of the operational approach is the direct execution of the software requirements specification. In this work, the JSD approach ("Jackson System Development") is considered as an operational approach and, as such, a language and a scheme are proposed that make possible the execution of JSD specifications. It is also implemented a system that analyzes specifications and allows their execution.

SUMÁRIO

LISTA DE FIGURAS

Pág.

CAPÍTULO 1

INTRODUÇÃO

O termo engenharia de software tem estado muito em voga nos dias de hoje. Quando ele foi introduzido, anos atrás, ele tinha um tom provocativo no sentido que indicava uma certa deficiência no mundo do computador, e a engenharia de software apontava uma solução.

O problema é que o software era produzido de modo artesanal e o desenvolvimento feito com pouca disciplina. O resultado era um software necessitando de constante manutenção, o que o tornava muito mais caro que o previsto.

A engenharia de software nasceu, então, com o objetivo de estabelecer e usar os princípios de engenharia para obter maior segurança e confiabilidade com menor custo. Entretanto o software não é um objeto físico, nenhum material o constitui ou envolve, o que proíbe a engenharia de software de ser simplemente uma cópia de outras áreas de engenharia (BAUER, 1975).

Vem daí o motivo da dificuldade em determinar uma divisão do trabalho de desenvolvimento de software em partes gerenciáveis como se ele fosse um processo industrial qualquer. Muitas divisões têm sido propostas, sob a denominação de "ciclo de vida do software".

O ciclo de vida incorpora todas as atividades necessárias para definir, implementar, testar, operar e manter um produto do software.

Um modelo de ciclo de vida, apresentado por Fairley (1985) consta das seguintes fases:

1) Análise - também chamada fase de especificação, inclui o entendimento do problema, um estudo da viabilidade observando objetivos e restrições, elaboração de uma estratégia de solução, determinação de um critério de aceitação e um planejamento do processo de desenvolvimento contendo a terminologia, os marcos, a estrutura gerencial, uma estimativa de custo, técnicas, ferramentas, etc. O resultado desta fase é a definição dos requisitos de software e hardware do sistema, contidos em um documento normalmente chamado Especificação de Requisitos.

2) Projeto - nesta fase é elaborado um projeto arquitetural onde são identificados os componentes do software bem como a relação entre eles, sendo imprescindível uma documentação de todas as decisões do projeto. Um projeto detalhado que defina como implementar o sistema e envolva: adaptação de partes de código já existentes, criação ou modificação de algoritmos, representação interna dos dados, definição do relacionamento entre os módulos, entre outros, também deve compreender esta fase.

3) Implementação - inclui a tradução das especificações de projeto na linguagem de programação adotada, a depuração, a documentação e os testes unitários do código fonte.

4) Teste - envolve a atividade de integração dos vários componentes para obter um produto que funcione, e a atividade de aceitação. A segunda envolve testes que demonstrem que o sistema implementado satisfaz os requisitos estabelecidos no documento de especificação de requisitos, elaborado na primeira fase do desenvolvimento.

5) Manutenção - inclui o aumento das capacidades do sistema, adaptação do mesmo a novos ambientes e a correção de eventuais erros.

O principal objetivo da divisão do trabalho de desenvolvimento do software é eliminar os erros de requisitos e de projeto de um produto antes de começar sua implementação. A correção de um erro de requisito durante a fase de teste é muito mais cara do que a correção do mesmo erro, durante a fase de análise (FAIRLEY, 1985).

A importância da primeira fase do ciclo de vida do software tem se evidenciado atualmente pois, correções de erros nessa fase tornarão o custo total do desenvolvimento significativamente menor.

Porém, apesar do ciclo de vida facilitar o gerenciamento do desenvolvimento ainda há um descontentamento com relação aos problemas crônicos do método tradicional. Novas idéias para o desenvolvimento de software têm surgido.

O Capítulo 2 apresenta alguns itens para uma boa especificação de requisitos e, em seguida, uma comparação entre o modelo convencional para o desenvolvimento de software e um novo modelo, chamado modelo operacional.

Várias linguagens de especificação operacionais já existem, entre elas está a notação gráfica do método "Jackson System Development" (JSD). O método de desenvolvimento de sistemas do Jackson (JACKSON, 1983) cobre todo ou quase todo o ciclo de vida do software. Uma especificação em JSD pode ser vista como uma rede de processos que se comunicam entre si. Desta forma, ela é em princípio, executável a menos de uma reconfiguração ou transformação da rede para rodar sobre um pequeno número de processadores reais ou virtuais. Uma descrição do método JSD é apresentada no Capítulo 3.

O Capítulo 4 descreve uma ferramenta de software, chamada JSD-tool, que auxilia a escrever e analisar especificações escritas segundo o método JSD.

A proposta deste trabalho é demonstrar a viabilidade dos modelos operacionais no que diz respeito a especificações executáveis, isto é, dada uma especificação escrita em uma linguagem formal, esta pode ser diretamente executada por um computador, com isso eliminando as outras etapas do ciclo de vida e consequentemente seus problemas. Para tanto, foi definida uma linguagem formal que permite escrever uma especificação segundo o modelo do Jackson e implementada uma máquina abstrata capaz de executar uma especificação escrita nesta linguagem.

As soluções e restrições assumidas para viabilizar a execução da rede de processos de uma especificação genérica em JSD são apresentadas no Capítulo 5. Neste capítulo, também são mostrados alguns detalhes do trabalho de desenvolvimento do Executor-JSD, assim chamada a ferramenta de engenharia de software construida para demonstrar que é possível executar uma especificação escrita em JSD.

O Capítulo 6 traz um exemplo imaginado por Jackson e descrito em seu livro (JACKSON,1983), sobre a automação de uma competição do jornal "Daily Racket". A descrição das regras da competição, das fases de especificação do método JSD até a versão escrita para o Executor-JSD, são discutidas neste capítulo.

CAPÍTULO 2

MOTIVAÇÃO PARA ESPECIFICAÇÕES EXECUTÁVEIS

2.1 - IMPORTÂNCIA DA FASE DE ESPECIFICAÇÃO

Estatísticas colhidas alguns anos atrás possibilitaram o conhecimento dos enormes e alarmantes custos de manutenção de grandes sistemas de software. Se a tendência continua, a indústria de software gastará mais tempo para manter programas velhos, mal estruturados e difíceis de modificar, do que para produzir novos sistemas.

Com o passar do tempo, o software tem se tornado cada vez mais poderoso. Em resposta a isso, as pessoas que o utilizam esperam mais de sua capacidade. Com esta crescente espectativa, vem a necessidade de aumentar a complexidade e/ou estender os padrões de programação. Esse aumento de complexidade e/ou da extensão dos padrões de programação, necessariamente implica em uma tarefa de validação mais extensa e complexa. Em um sistema onde o cumprimento dos padrões é manual, aumenta a probabilidade de ocorrência de erros, podendo o software até mesmo ser perdido (HOWLEY, 1983).

Além disso, tem sido mostrado que a possibilidade de mudanças introduzirem erros adicionais aumenta com o crescimento da complexidade do sistema. Mas, o perigo real está na dependência cada vez maior da sociedade em tais sistemas, principalmente no que diz respeito a área de sistemas bélicos (YEH, 1984).

Isto vem fazendo com que, desde 1968, a Engenharia de Software se preocupe com o software como um produto e forneça contribuições a todas as fases do seu ciclo de vida. A necessidade e importância de se definir os requisitos do software tem ganho importância cada vez maior e pode ser comparada às mesmas necessidades encontradas por outras engenharias. Por exemplo, Engenharia Civil não inicia a construção de um prédio antes de levantar as necessidades dos moradores e interesses dos construtores, definindo tanto as características internas como as cataracterísticas externas do prédio levando em conta as condições ambientais do terreno onde se dará a construção, etc.

O documento de especificação de requisitos, no desenvolvimento de um sistema de software é a base da comunicação entre **compradores, usuários, analistas** e **programadores** (os termos em negrito estão definidos no glossário do Apêndice A), e se ele não representar um consenso destes grupos o projeto provavelmente não terá sucesso. O que geralmente acontece, é que o comprador não entende o projeto de software e o processo de desenvolvimento bem o suficiente para escrever uma especificação de requisitos útil aos analistas e programadores. Por outro lado, os analistas, geralmente, não entendem o problema do comprador nem sua área de interesse o bastante para especificar os requisitos do sistema de forma satisfatória aos compradores.

Dadas as dificuldades e insucessos com os documentos de especificação, pesquisadores têm-se voltado para esta área da engenharia de software, na tentativa de auxiliar o desenvolvimento de sistemas de software, barateando o custo e assegurando maior confiabilidade no produto (BALZER, 1979).

O Software Engineering Technical Committee of the IEEE Computer Society juntamente com o Standards Coordinating Committees of the IEEE Standards Board, com o objetivo de estabelecer uma base para acordos entre usuários e fornecedores do software, reduzir o esforço de desenvolvimento, estabelecer uma base para validação, facilitar a transferência e outros, elaborou um guia para se escrever uma especificação de requisitos de software (IEEE, 1984).

Este guia descreve o conteúdo e as qualidades necessárias para uma boa Especificação de Requisitos de Software, cujos principais itens a serem observados são descritos a seguir:

a) Não ambígua: todos os requisitos devem ter somente uma interpretação. Isto requer que cada característica do produto final seja descrita usando um único termo. Termos que normalmente têm múltiplos significados ou podem causar dúvida ao leitor, devem ter uma definição precisa do seu significado contida em um glossário.

b) Completa: deve incluir a definição de todos os requisitos significativos, relatar a funcionalidade, o desempenho, as restrições de projeto, os atributos ou interfaces externas; definir as respostas do software para todas as classes de dados de entrada em todas as situações inclusive para as condições de erros; manter todas as tabelas, figuras e diagramas rotulados e conter uma definição dos termos e unidades de medidas. Uma especificação dita completa não pode conter a frase **a ser definida** (ASD).

c) Verificável: todo requisito da especificação deve ser verificável. Um requisito é verificável se, e somente se, existe um processo efetivo com o qual uma pessoa ou máquina pode verificar se o resultado do software está de acordo com o requisito.

d) Consistente: não deve existir conflito entre os requisitos individuais descritos. Três tipos de conflitos devem ser evitados: conflito entre dois ou mais requisitos que descrevem o mesmo objeto do mundo real; conflito entre as características do mundo real especificadas; conflito lógico e temporal entre duas ações especificadas.

e) Modificável: a estrutura e o estilo da especificação devem ser tais que toda mudança necessária nos requisitos possa ser feita fácil, completa e consistentemente. Esta característica requer que a especificação tenha uma organização coerente e fácil de usar e não seja redundante, isto é, que um mesmo requisito não apareça em mais de um lugar.

f) Facilmente rastreável: a origem de cada requisito deve ser clara e facilitar a referência a cada requisito no desenvolvimento futuro ou aumento da documentação.

g) Usável durante a fase de manutenção e operação: a especificação deve direcionar as necessidades das fases de operação e manutenção incluindo a eventual substituição do software.

A maneira mais usual e fácil de criar uma especificação de requisitos de software é descrevê-la em linguagem natural. Mas, devido a riqueza das linguagens naturais, o documento tende a ser impreciso não satisfazendo muitas das características acima.

Com o aumento efetivo de custo e com um número limitado de pessoas produzindo software, percebeu-se uma necessidade clara de investigar um paradigma de software baseado na automação. Na 4a. Conferência Internacional sobre Engenharia de Software, ocorrida em Munique (Alemanha - 1979), Leon Stucki, então chairman, proferiu as seguintes palavras: "paradoxally, software engineering has done an excellent job of automating everyone else's job except its own"[1], as quais sintetizam perfeitamente um novo direcionamento da pesquisa na área de engenharia de software (URBAN, 1985).

Várias metodologias para o desenvolvimento do software vêm despontando. Muitas novas alternativas puderam ser organizadas em uma estrutura chamada "modelo operacional". Em oposição a este novo modelo denominou-se os demais de "modelo convencional" (ZAVE, 1984).

No **modelo convencional**, analistas de sistemas, durante a fase de especificação, formulam e definem um sistema em linguagem natural (pois não existe uma linguagem formal conveniente). Esta especificação trata o sistema como uma **"caixa preta"**, descrevendo todas as características do seu comportamento externo (o que) e nenhuma característica da estrutura interna (como) que gerará o comportamento externo. Neste modelo a decomposição da complexidade baseia-se no princípio descendente (top-down) da decomposição hierárquica de "caixas pretas". Sob este modelo reside quase toda teoria de engenharia de software.

O modelo convencional apresenta falhas fundamentais que exacerbam o problema de manutenção. A manutenção, neste modelo é baseada no código, ou seja, na implementação que, por sua vez, é difícil de entender e modificar.

---

[1] "paradoxalmente, a engenharia de software tem feito um excelente trabalho na automatização dos trabalhos de todos, exceto o seu próprio".

O **modelo operacional** é um paradigma para o desenvolvimento de software que nasceu do descontentamento com o modelo convencional. Neste modelo a especificação é escrita em uma linguagem formal, executável (gerando protótipos que, ao serem executados, pareçam reais), independente da implementação (no sentido que suas características possam ser realizadas por uma grande variedade de configurações de recursos), e aplicável a muitos tipos de sistemas.

Uma comparação mais detalhada entre os dois modelos pode facilitar o entendimento das novas propostas sobre os métodos operacionais.

2.2 - COMPARAÇÃO ENTRE O MODELO CONVENCIONAL E O MODELO OPERACIONAL

O objetivo de qualquer método de desenvolvimento de software é solucionar um dado problema através do uso de um computador.

Os dois modelos podem ser vistos como uma sequência de decisões. A cada nova decisão pode haver necessidade de retificação nas decisões anteriores. As diferenças entre os dois modelos se manifestam logo no incio. Para facilitar a comparação entre os modelos eles foram subdivididos em três grandes fases. A Figura 2.1 e a Figura 2.2 mostram estas fases de desenvolvimento dos modelos convencional e operacional, respectivamente, e servem como uma estrutura visual para a comparação.

As três fases do modelo convencional são as seguintes:

1) **Requisitos:** o sistema é definido segundo seu comportamento externo. O segundo retângulo totalmente fechado da figura 2.1, significa que não se conhece internamente a forma como o sistema recebe e manipula as entradas para produzir as saídas desejadas. O sistema é uma caixa preta. O documento gerado nesta fase é quase sempre escrito em linguagem natural e, portanto, informal;

2) **projeto:** a estrutura interna do sistema é definida, geralmente, como uma decomposição de módulos de código, onde cada módulo é responsável por uma função. A estrutura interna de cada módulo ainda é uma "caixa preta". Do ponto de vista organizacional, cada pessoa pode ficar responsável pela implementação e teste de um ou mais módulos. Um projeto mais substancial inclui alocação preliminar de recursos, plano de testes e outros, baseados nos recursos de máquina disponíveis;

3) **implementação:** o projeto é passado para uma linguagem de programação. São estabelecidos os mecanismos internos de cada módulo e seus desempenhos de acordo com a linguagem de programação e o ambiente de software/hardware disponíveis.

O modelo convencional coloca grande ênfase na separação dos requisitos do comportamento externo e da estrutura interna, não permitindo interrelacionamento entre as fases do desenvolvimento. Os mecanismos de alto nível são totalmente independentes dos mecanismos de baixo nível, determinados durante a implementação.

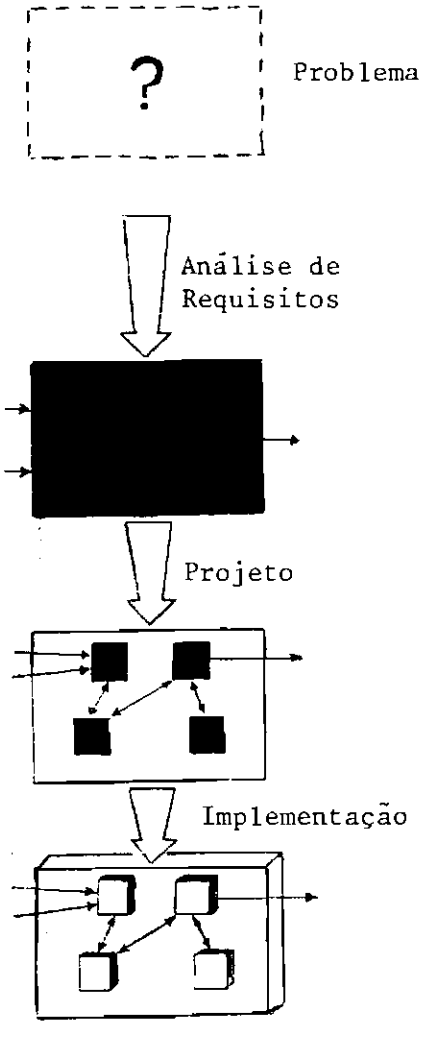


Fig. 2.1 - Modelo Convencional

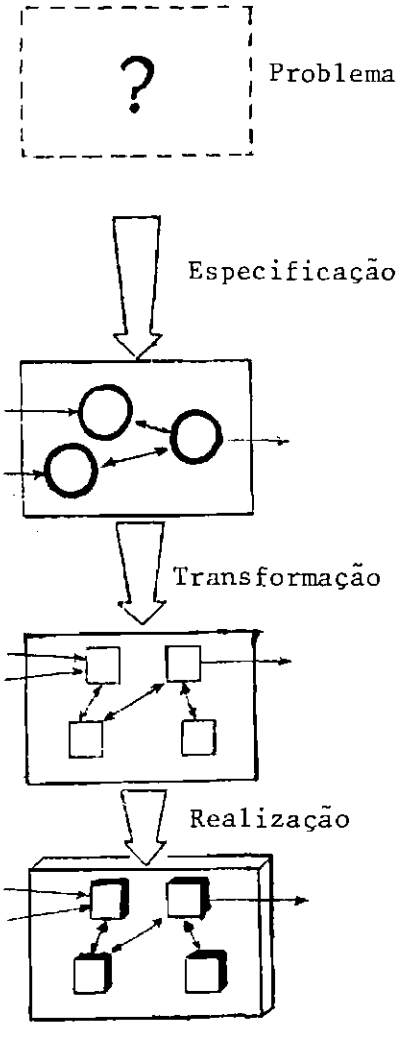


Fig. 2.2 - Modelo Operacional

FONTES: ZAVE (1984) p. 105 e 108 ; ZAVE (1983) p.2

As três fases do modelo operacional são as seguintes:

1) **Especificação:** o sistema é formulado em termos de estruturas independentes da implementação, capazes de gerar o comportamento externo do sistema. A linguagem usada é formal e, portanto, capaz de ser executada através do auxílio de um interpretador conveniente. As estruturas da especificação são independentes da configuração e da alocação de recursos da máquina podendo ser implementada em uma grande variedade delas. Tanto as estruturas como os mecanismos usados na linguagem de especificação são derivados somente do problema a ser resolvido, sem considerar as características de programação;

2) **transformação:** a especificação é transformada mantendo o comportamento externo do sistema. Apenas são inseridos ou alterados alguns mecanismos que produzem as saídas, tornando-os mais próximos da implementação. Um tipo de transformação troca os mecanismos modificáveis da especificação para mecanismos que atendam um balanceamento de desempenho e recursos de implementação tais como processadores, memória, canais de comunicação, etc. Outras transformações podem, não simplesmente modificar mas, também, introduzir representações explícitas de recursos de implementação, ausentes na especificação original. Cabe ressaltar que muitas decisões aqui podem trazer problemas de desempenho quando automatizadas, sobrando atividades que dependam de decisões humanas, como por exemplo, escolhas de algoritmos;

3) **realização:** a especificação transformada é mapeada diretamente na linguagem de programação. Se o ambiente onde será implementado o sistema não comportar as facilidades necessárias, a realização cria uma máquina virtual sobre a qual a especificação transformada possa ser executada.

Para a realização do modelo operacional, Pamela Zave (ZAVE, 1982) e Michael Jackson (JACKSON, 1983; CAMERON, 1986), em particular, propõem que seja elaborado, inicialmente, um modelo do mundo real independente das funções desejáveis do sistema. Algumas das várias razões, para elaborar o modelo previamente independente das funções, são que:

a) O modelo é mais explícito e serve como base para comunicação entre compradores e analistas;

b) o modelo define os termos e estabelece implicitamente um conjunto de possíveis funções;

c) o modelo é mais estável que as funções;

d) o modelo é independente das funções;

e) muitas das restrições de desempenho são obtidas mais naturalmente, pois a capacidade funcional do sistema está implícita no modelo.

As características de cada um dos modelos são apresentadas na Tabela 2.1. A cada item da tabela precede uma indicação (v) se a característica corresponde a uma vantagem e (d) se a característica corresponde a uma desvantagem do modelo. O conteúdo da tabela completa a comparação entre os dois modelos.

TABELA 2.1

CARACTERÍSTICAS DOS MODELOS CONVENCIONAL E OPERACIONAL SEPARADAS EM VANTAGENS(v) E DESVANTAGENS (d)

Modelo Convencional
v- a especificação de requisitos é informal, escrita em linguagem natural, portanto pode ser entendida, em princípio, por qualquer pessoa;
v- apresenta marcos organizacionalmente úteis sob o ponto de vista gerencial;
d- não conta com recursos para geração de um protótipo, logo, o usuário demora a ver um resultado (*);
d- a validação do sistema é feita sobre o código, após a implementação, sendo, portanto, lenta e imprecisa;
d- a fase de implementação é geralmente manual;
d- grande parte da documentação é feita manualmente e portanto, muitas decisões de projeto podem ser perdidas;
d- pode haver problemas de ausência de pessoal capacitado para desenvolver sistemas muito grandes;
d- não conta com facilidades para a interação entre vários especialistas, de áreas diferentes, para o desenvolvimento de sistemas grandes;

(continua)

Tabela 2.1 - Continuação.

Modelo Convencional
d- durante a fase de requisitos o sistema é uma "caixa preta", o que pode tornar os requisitos um tanto obscuros; d- o desenvolvimento é lento havendo muita troca de pessoal até a conclusão do sistema; d- a decomposição do sistema é do tipo descendente (top-down) e portanto, arriscada.
Modelo Operacional
v- a especificação de requisitos é formal, escrita em uma linguagem formal, portanto, mais precisa; v- um protótipo do sistema pode ser criado automaticamente (*); v- a especificação é executável permitindo que a validação do sistema seja feita sobre o protótipo, logo após a fase de especificação, assim o usuário tem um modelo do sistema mais rapidamente; v- a fase de realização, onde é feita a implementação, é auxiliada por computador e, portanto, a correção é garantida, desde que a especificação esteja correta; v- o desenvolvimento pode ser documentado automaticamente, dado que a especificação é executável;

(continua)

Tabela 2.1 - Conclusão.

Modelo Operacional
v- a interação entre especialistas de áreas diferentes, durante o desenvolvimento de sistemas grandes é facilitada pois todos devem "falar a mesma linguagem" e por haver ferramentas automatizadas;
d- a automação do desenvolvimento pode trazer problemas sociais com a redução do número de pessoas necessárias ao desenvolvimento;
d- pode ser difícil executar especificações com performance adequada;
d- para entender os requisitos é preciso aprender a linguagem de especificação, o que provoca uma resistência, principalmente dos compradores, em adotarem tais modelos;
d- os aspectos gerenciais são totalmente inexplorados, no momento;
d- a implementação transformacional é uma tecnologia nova e mal desenvolvida, podendo ser difícil direcioná-la;
d- um problema com especificação operacional é que elas podem executar muito lentamente alguns testes e demonstrações, não correspondendo à realidade.

(*) O conceito de protótipo soma novas dimensões a possibilidade de participação do usuário no desenvolvimento de um sistema. Um protótipo é considerado um modelo 'ad hoc', implementado e em funcionamento, usado para auxiliar a tomada

de decisões quando o sistema desejado ainda não é bem conhecido. O uso de um protótipo durante a fase de análise e projeto torna estas fases do ciclo de vida mais dinâmicas, verificando constantemente o conhecimento de analistas e usuários sobre o problema, evitando desentendimentos entre eles e garantindo a qualidade da análise. O objetivo de criar protótipos é assegurar o desenvolvimento eficiente de um sistema real atendendo todos os requisitos (DEARNLEY, 1983). No modelo operacional, a especificação pode ser usada como um protótipo desde que seja executável. No modelo convencional, o protótipo é produzido via uma interação do usuário no ciclo todo, mas não há um guia de como produzir um protótipo mais rapidamente que o produto.

No modelo operacional as especificações de requisitos são executáveis. Uma especificação executável é um modelo de um sistema que pode ser executado (sob um executor conveniente) e simular o comportamento do sistema especificado.

A diferença entre uma especificação executável e um modelo de simulação é que o modelo de simulação somente reproduz aspectos particulares do comportamento do sistema simulado, enquanto a especificação executável pretende representar o sistema de uma forma compreensível e independente da implementação, simplesmente reproduzindo seu comportamento.

A diferença entre uma especificação executável e um programa é que um programa deve ser compilável automaticamente gerando um código relativamente otimizado. Enquanto que uma especificação deve ser uma abstração de todas as possíveis formas de implementação (ZAVE, 1987).

Uma especificação executável deve ser compreensível e modificável em todos os termos do domínio de aplicação, bem como livre das decisões que restringem a implementação.

As linguagens de especificação executáveis (ou operacionais) são projetadas para atender os objetivos, suposições e expectativas do modelo operacional. O Capítulo 3 ilustra um modelo operacional, e a linguagem de especificação definida para ele é uma linguagem de especificação executável pois foi possível construir-se um executor para a especificação produzida segundo a linguagem do modelo.

CAPÍTULO 3

JSD - JACKSON SYSTEM DEVELOPMENT

3.1 - CONSIDERAÇÕES GERAIS

O JSD cobre todo (ou quase todo) o ciclo de vida do software, porém não mantém rigorosamente a divisão apresentada no primeiro capítulo. Primariamente, o método apresenta duas grandes fases: especificação e implementação. As outras fases do ciclo de vida ficam diluidas dentro destas duas divisões. Por exemplo, a fase de projeto é toda absorvida pela fase de implementação do JSD, como veremos a seguir.

As duas fases foram propositalmente escolhidas com o objetivo de encorajar e clarificar uma divisão entre as pessoas cujo interesse primário reside no uso do sistema (compradores e usuários) e as pessoas cujo interesse maior reside na computação (analistas e programadores). O comprador/usuário tem grande participação na fase de especificação. Ele deve cooperar com conceitos e notações para que analistas possam entender o funcionamento da área de interesse do **mundo real**, com a qual o sistema está relacionado, e para que a especificação possa ser expressada claramente.

Na fase de especificação, a idéia central é a elaboração de um **modelo** do mundo real, antes da definição de suas funções. (Isto quebra a tradição dos modelos convencionais onde o primeiro passo é a definição dos requisitos funcionais e das entradas e saídas). O modelo deve agir e responder da mesma forma como agem e respondem os objetos reais. O sistema é considerado um tipo de simulação do mundo real.

A descrição abstrata do mundo real é feita em termos de **entidades** e **ações** (dois conceitos definidos no JSD). As entidades correspondem a objetos e as ações a eventos do sistema. É importante lembrar que as entidades e ações são identificadas no contexto do mundo real, fora do contexto do computador, de acordo com a visão do comprador e do usuário, evitando a visão excessivamente técnica do analista. Uma vez identificadas as entidades, estas serão caracterizadas por um conjunto ordenado de ações. A escolha da ordem das ações impõe restrições ao sistema.

As entidades isoladas não caracterizam um modelo, elas são apenas os elementos que compõem o modelo. O significado de entidade no JSD está fortemente associado ao conceito de **processo modelo**, pois cada entidade dá origem a um processo sequencial.

Desta forma, as especificações em JSD constituem, principalmente, uma rede de processos que se comunicam por troca de mensagens e por inspeção, somente de leitura, a dados de outros processos modelos. Novos processos são incluidos, através de conexões deles ao modelo (se necessário), quando da introdução das funções.

Com esta configuração, a especificação é, em princípio executável. Existe, frequentemente, um problema entre o número e as características dos processos na especificação e o número e as características dos processos que podem ser convenientemente executados nas condições da máquina e do sistema operacional disponíveis. A fase de implementação se preocupa em transformar a especificação em um sistema convenientemente executável ajustando os processos e dados aos processadores e memória disponíveis. O objetivo é diminuir a distância entre a especificação e a programação.

Mais detalhadamente, o método de desenvolvimento JSD está dividido em seis passos, assim denominados:

1) Ações e Entidades: neste passo, o analista define a área de interesse do mundo real listando as entidades e as ações com as quais o sistema está relacionado.

2) Estrutura da Entidade: neste passo, as ações executadas ou sofridas por cada entidade são organizadas em ordem de ocorrência. Cada entidade é representada através de um diagrama.

3) Modelo Inicial: neste passo, a descrição da realidade, em termos de entidades e ações, é realizada em um modelo de processos e conexões entre o modelo e o mundo real.

4) Função: neste passo, são especificadas as funções que produzirão as saídas do sistema, podendo ser somados novos processos ao modelo.

5) Sincronismo do Sistema: aqui, o analista considera alguns aspectos de escalonamento dos processos, os quais podem afetar a correção e temporização das saídas funcionais.

6) Implementação: neste passo, são considerados o hardware e o software onde o sistema será executado conveniente e eficientemente.

Os dois primeiros passos estão relacionados com a descrição abstrata do mundo real em termos de processos sequenciais. Após a realização do terceiro passo tem-se o modelo ou simulação da realidade. O quarto e quinto passos estão relacionados com as definições dos requisitos funcionais: as funções propriamente ditas e o tempo para as respostas do sistema. No sexto passo são feitos os ajustes ao modelo para que ele possa ser executado e assim, é efetivada a implementação do sistema.

Uma das grandes vantagens do tipo de modelo construido no JSD é a facilidade com que ele pode ser transformado em um conjunto de programas capazes de serem executados em computador. Outra vantagem, é sua grande aplicabilidade: o único requisito exigido para aplicar o método é que as dimensões de tempo do sistema a ser desenvolvido sejam claramente identificadas.

O JSD não incorpora atividades associadas com o desenvolvimento do sistema, tais como: seleção, planejamento e gerenciamento do projeto, análise de custo/benefício, procedimento para instalação, aceitação e outros.

Os passos do JSD são descritos com mais detalhes na Seção 3.2 e exemplificados no Capítulo 6. O Apêndice A traz uma definição dos principais termos usados na descrição deste método.

3.2 - OS PASSOS DO JSD

3.2.1 - PASSO 1: AÇÕES E ENTIDADES

No passo ações e entidades, a principal preocupação é definir do que se trata o sistema que será desenvolvido.

Nesta fase, o analista, responsável pelo desenvolvimento, olha o sistema de fora e procura identificar as entidades nas pessoas, coisas e organizações, observando como elas agem, e quais as suas principais ações tendo sempre em mente os objetivos gerais do sistema.

Após a observação do mundo real fora do sistema, deve ser elaborada uma lista de entidades e uma lista de ações correspondentes a cada entidade. Necessáriamente a lista é seletiva. A escolha das entidades e ações incluidas no modelo irão estabelecer o escopo do sistema.

Alguns critérios são estabelecidos para a escolha das entidades. Uma entidade deve obedecer a todos os requisitos abaixo:

a) Executar ou sofrer ações em uma ordem de tempo bem determinada. Por exemplo, "data" **não** deve ser uma entidade;

b) existir no mundo real fora do sistema e não ser meramente uma parte do sistema ou um produto dele. Por exemplo: "registro de erro" **não** deve ser uma entidade;

c) ser individual ou única e, se existirem mais de um tipo da mesma entidade, ela deve poder ser unicamente denominada;

d) não ser inerte.

Outros critérios são estabelecidos para a escolha das ações de uma entidade. Assim, uma ação deve:

a) Ser considerada em um único instante particular do tempo, não podendo se estender por um período de tempo;

b) acontecer no mundo real fora do sistema e não ser meramente uma ação do próprio sistema. Por exemplo: "imprimir mensagem de erro" não é uma ação permitida;

c) ser atômica, não podendo ser decomposta e originar outras ações.

A fim de tornar a lista de ações mais precisa e compreensível, deixando claro o que acontece no mundo real, é fornecida uma descrição informal e uma lista de atributos para cada ação. Durante o desenvolvimento do sistema a lista de atributos das ações vai se extendendo, particularmente no passo de Função. O que não acontece com a lista de entidades que é totalmente definida neste passo do desenvolvimento.

Entretanto, tanto a lista de entidades como a lista de ações poderão ser alteradas durante os passos da Estrutura da Entidade e do Modelo Inicial, significando uma volta ao passo inicial. No primeiro passo estas listas são mais um tentativa do que algo definitivo.

Cada entidade é unicamente caracterizada por um conjunto ordenado de ações. As entidades escolhidas darão origem, futuramente, a processos sequenciais.

A elaboração das listas de ações e entidades encerra este passo do método JSD, o qual é elaborado por analistas e, principalmente, pelos compradores e usuários que possuem maior conhecimento do mundo real. Estas listas farão parte do glossário de termos do sistema e são as entradas para o passo seguinte.

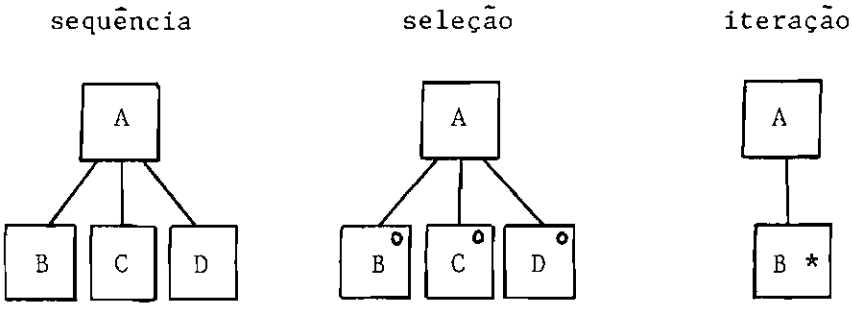
3.2.2 - PASSO 2: ESTRUTURA DA ENTIDADE

No passo da estrutura da entidade, as ações de cada entidade são organizadas de acordo com sua ordem de ocorrência no tempo e representadas em uma notação gráfica denominada diagrama de estrutura.

O diagrama de estrutura é usado para facilitar a visualização do comportamento da entidade. Esta notação evita obscuridade e imprecisão na especificação e é fácil de aprender.

Cada entidade, mais suas ações, dão origem a um único diagrama. Para a construção do diagrama é recomendada a expansão do mesmo levando em conta todo o tempo de vida da entidade no mundo real.

Os diagramas de estrutura são árvores construidas com os três componentes clássicos de programação estruturada: sequência, seleção e iteração, como mostrado na Figura 3.1.



sequência	seleção	iteração
A - componente de sequência, B,C,D - ações que ocorrem nesta ordem	A - componente de seleção, B,C,D - ações, apenas uma delas ocorre	A - componente de iteração, B - ação repetida uma ou mais vezes

Fig. 3.1 - Notação das estruturas que podem compor um diagrama de estrutura.

Nas árvores, o nó raiz recebe o nome da entidade e as folhas correspondem às ações da mesma.

Uma sequência é identificada por caixas sem marcas para os filhos. O significado de uma sequência é que, para cada instância de sequência, ocorre somente uma instância de cada filho, da esquerda para a direita no diagrama. Uma sequência pode ter qualquer número de filhos, sendo permitido um único. Mas não faz sentido uma sequência sem filhos. Todos os filhos devem estar explícitos no diagrama.

O caso de duas ou mais ações nunca ocorrerem ao mesmo tempo, é representado por uma seleção. Na seleção, sempre deve haver mais de um filho, não importando a ordem entre eles. Em alguns casos, quando folha, um filho pode ser nulo ou vazio. Uma seleção é indicada através de um pequeno círculo no quadrante superior direito da caixa dos filhos.

Uma iteração é usada quando o número de ocorrências de uma dada ação é desconhecida na fase de especificação. O significado da iteração é que, para cada instância de iteração, o filho ocorre zero ou mais vezes. O número de ocorrências não faz parte da notação. As ocorrências das partes de uma iteração são sequencialmente ordenadas. Uma iteração é indicada com um asterisco no quadrante superior direito da caixa dos filhos.

Os diagramas criados neste passo não expressam concorrência interna, pelo contrário, eles demonstram a sequência de ações de uma entidade.

As estruturas de entidades têm um único propósito: especificar as restrições sobre as possíveis ordenações das ações da entidade no mundo real. Não devem, pois, ser elaboradas com os requisitos funcionais em mente.

Algumas entidades podem surgir neste passo, derivadas de certas ações de uma outra entidade. Uma entidade derivada da estrutura de outra entidade é chamada entidade marsupial. Por exemplo, se o usuário de uma biblioteca pode retirar vários tipos de livros com diferentes regras para retirada e devolução, o comportamento do usuário não pode ser mostrado em um único diagrama. É extraída uma nova entidade EMPRÉSTIMO da estrutura USUÁRIO.

Finalizando o passo 2, tem-se a especificação do diagrama de estrutura e, por conseguinte, a ordem das ações de cada entidade. Com isso, pode-se iniciar o passo de modelamento do sistema.

3.2.3 - PASSO 3 : MODELO INICIAL

No passo do modelo inicial tem início a construção da parte dinâmica do sistema. O primeiro caminho a seguir é especificar um conjunto de processos sequenciais que modelem o comportamento das entidades especificadas no passo anterior.

Os processos sequenciais são extraídos diretamente das estruturas das entidades. Cada diagrama de estrutura é transformado em um processo sequencial capaz de ser executado em um computador. Para isso a transformação se dá de uma notação gráfica para uma notação textual. O processo sequencial derivado da estrutura da entidade é chamado **processo modelo**. Desta forma, cada processo modelo simula exatamente o comportamento de cada entidade.

As transformações aplicadas na estrutura da entidade, para se obter o processo modelo correspondente, são:

1) o nome do nó raiz passa a ser o nome do processo;

2) as folhas do diagrama são representadas por seus nomes seguidos de ponto e vírgula;

3) em um nó sequência, a ordem das partes da esquerda para a direita é mantida de cima para baixo entre o começo e o fim da estrutura textual da sequência. O nó sequência ilustrado na Figura 3.1, da Seção 3.2.2, teria a seguinte notação textual:

```
A seq
        B;
        C;
        D;
A end
```

4) em um nó seleção, a primeira parte segue o começo da seleção e as outras partes são precedidas pela palavra **alt** (alternativa). O nó seleção ilustrado na Figura 3.1 teria a seguinte notação textual:

```
A sel ( cond-B )
      B;

A alt ( cond-C )
      C;

A alt ( cond-D )
      D;

A end
```

5) em um nó iteração, a única parte é representada entre o começo e o fim do texto da iteração. O nó iteração ilustrado na Figura 3.1 teria a seguinte notação textual:

```
A iter ( cond-B )
      B;
A end
```

Após a transcrição direta das construções citadas, devem ser introduzidas as condições nas diversas iterações e seleções.

Para assegurar que o comportamento do modelo reflete os eventos do mundo real, é necessário conectar os processos modelos, direta ou indiretamente, ao comportamento do mundo real.

Existem duas formas de conexão de processos no JSD, a saber: conexão por sequência de dados e conexão por vetor de estado.

Na conexão por sequência de dados, um processo escreve uma sequência de dados, consistindo de um conjunto ordenado de mensagens ou registros e o outro processo da conexão lê esta sequência.

Na conexão por vetor de estado, um processo lê um conjunto de variáveis locais pertencentes, internamente, ao outro processo da conexão.

Os dois tipos de conexão são usados tanto para ligar os processos modelos ao mundo real como para ligar os processos modelos a processos internos ao sistema.

O texto da estrutura, agora, pode ser acrescido de operações de leitura e escrita a dados, bem como, variáveis de controle a fim de representar a viabilidade das comunicações entre os processos conectados.

Com o objetivo de mostrar quais processos estão diretamente ligados e qual o tipo da conexão existente entre eles, o modelo é representado por uma outra notação gráfica, chamada **Diagrama de Especificação do Sistema - DES** ('System Specification Diagram - SSD').

O DES é composto por retângulos que representam os processos; círculos que representam as conexões por sequência de dados; losângulos que mostram as conexões por vetor de estado; e setas que indicam a direção do fluxo de informação na conexão. Algumas vezes, é preciso representar a multiplicidade de processos ou dados, por exemplo, uma conexão pode ligar um processo a outro ou muitos processos a um único, ou um único processo a muitos ou ainda muitos processos de um tipo a muitos processos de outro tipo. Isso é feito com uma barra dupla desenhada sobre a seta, no lado onde há muitos processos.

No DES tanto os processos como as conexões são devidamente denominadas. Neste passo, o DES é composto por processos de dois níveis: nível zero e nível um. Os processos de nível zero recebem o nome da entidade seguido de **-0**. Esses processos representam o objeto do mundo real que fornece as entradas ao sistema e não um processo de máquina. Já os processos de nível um correspondem aos chamados processos modelos, aqueles que simulam o comportamento da entidade dentro do computador. Eles são nomeados com o próprio nome da entidade seguido de -1.

Na tarefa de realizar a simulação do comportamento das entidades do mundo real, o JSD estabelece como forma padrão de conexão, entre o mundo real e a entidade, a conexão por sequência de dados. A entidade do mundo real representada como um processo do nível 0 produz uma mensagem após cada ação executada. A sequência de mensagens vai sendo armazenada em um "buffer", o qual é a entrada para o processo modelo (de nível 1). Desta forma, os processos de nível 0 têm sempre uma operação de escrita após cada ação e os processos de nível 1, uma operação de leitura antes de cada ação. A operação de leitura segue a regra "leia à frente", pois a ação a ser tomada pelo processo modelo, depende da mensagem da entidade do mundo real, já que o primeiro modela fielmente o segundo.

Com o objetivo de facilitar a sincronização entre os processos, a conexão por sequência de dados comporta-se como um buffer de mensagens obedecendo o algorítmo de substituição primeiro-que entra, primeiro-que-sai ('First In, First Out - FIFO'). Nesta fase, o buffer é considerado ilimitado. Também não é considerada a velocidade de execução de cada processo.

O sincronismo entre os processos envolvidos segue o mecanismo tradicional do "produtor/consumidor" (HANSEN, 1973): se o buffer está cheio, o processo que escreve espera até que o processo leitor continue sua leitura, pois a cada leitura um espaço é liberado no buffer e assim, o processo escritor pode continuar sua execução. Da mesma forma, quando o buffer está vazio, o processo leitor não pode continuar sua execução, ficando bloqueado até que lhe seja enviada uma mensagem.

A conexão através de vetores de estados é usada quando se deseja uma outra condição de conexão. Este tipo de conexão também pode ser usado para ligar processos do nível 0 aos processos modelos. Aqui, a iniciativa de comunicação sempre parte do processo modelo. Este possui uma operação especial para ler o vetor de estado do outro, chamada "getsv". Após a execução de "getsv" o processo modelo fica com uma cópia, em sua área, do vetor de estado do outro processo. Os valores do vetor de estado do proceso do mundo real não são alterados por "getsv".

A sucessão de valores obtidos pelo processo modelo, através de "getsv", depende da velocidade de execução dos processos. Neste tipo de conexão os processos envolvidos nunca ficam bloqueados. Para que haja consistência nos valores obtidos, deve haver uma sincronização temporal entre os processos envolvidos. Entretanto, essa preocupação é deixada para a fase de implementação.

O JSD conta com alguns detalhes para a elaboração do modelo destinados a atender necessidades particulares, por exemplo:

1) um processo pode ter múltiplas entradas e múltiplas saídas, as quais podem ter velocidades diferentes de chegada ou partida respectivamente. As mensagens das sequências de dados devem ser absorvidas de acordo com regras determinadas na especificação, mantendo uma ordem de absorção que melhor satisfaça ao sistema. O tratamento de múltiplas entradas, por um único processo, recebe o nome de fusão fixa ('fixed merge');

2) alguns processos podem ter estruturas de dados complexas, sendo necessário descrever a estrutura na documentação da especificação. A notação usada para descrever estruturas de dados é a mesma notação de diagramas de estruturas usada na descrição dos processos;

3) se um processo, que recebe mensagens de duas sequências de dados diferentes, deve manter a ordem de chegada das mensagens, a fusão fixa das entradas não resolve o problema, pois a velocidade e a ordem de chegada das mensagens é aleatória e imprevisível. Para resolver esta situação, o JSD apresenta um esquema chamado **fusão forçada** (rough merge) das entradas. O processo que absorve as sequências recebe as mensagens como se elas chegassem por uma única entrada. No DES a fusão forçada é indicada com duas ou mais sequências de dados cujas setas se unem no lado do processo que as absorve. A Figura 3.2 ilustra uma fusão forçada de duas sequências de dados E e F que são absorvidas pelo processo PROC, seguindo a notação usada no DES. A fusão forçada ilustrada na Figura 3.2 seria referenciada como E&F na notação textual do processo PROC.

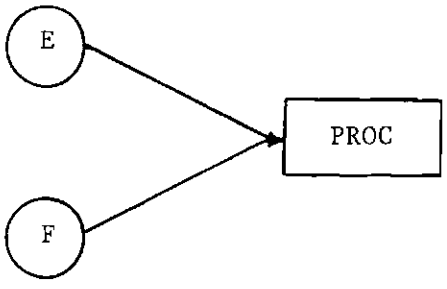


Fig. 3.2 - Notação usada para indicar uma fusão forçada.

5) outras vezes, é preciso modelar uma ação que seja executada a partir de um dado instante de tempo ou mesmo por um período de tempo. Para isso o processo precisa reconhecer a chegada de um ponto particular no tempo. No JSD há um tipo de entrada, denominada **marcador de tempo** (Time Grain Marker - TGM), que sinaliza, ao sistema, a chegada do instante de tempo. Os marcadores de tempo quase sempre fazem parte de uma fusão forçada.

Concluindo o passo de modelagem ou simulação do mundo real, pode-se observar que o sistema não tem saídas e as entradas são simplesmente mensagens que sinalizam a ocorrência de eventos do mundo real. Neste ponto está pronta a base para a definição das funções a serem introduzidas no sistema.

3.2.4 - PASSO 4 : FUNÇÃO

No passo das funções, como seu próprio nome sugere, são especificadas as funções do sistema.

Os três passos anteriores proporcionam o modelo ou simulação que é a base para a introdução dos requisitos funcionais. Uma combinação de eventos que ocorre no mundo real é modelada através de uma combinação de eventos dentro do sistema. As funções são introduzidas ao modelo de forma que elas produzam uma saída quando ocorrer uma combinação específica de eventos.

Uma vantagem importante do JSD é que cada função é modelada independentemente da outra pois, em geral, os requisitos funcionais baseados em um modelo comum são altamente independentes. Isso permite que várias pessoas trabalhem simultâneamente no passo de Função e na manutenção futura. Permite também, que novas funções sejam somadas sem interferir nas já definidas.

As funções são somadas ao sistema na forma de **processos funções.** Os processos funções recebem entradas dos processos modelos e produzem saídas em impressoras, terminais, discos, e outros equipamentos. Estes processos vão sendo introduzidos ao DES (Diagrama de Especificação do Sistema) obedecendo a mesma notação usada para os processos modelos.

A seguir são discutidas algumas formas de introduzir as funções ao modelo do sistema.

Caso a combinação de eventos seja simples e a computação da função seja direta, as operações da função podem ser diretamente embutidas na estrutura do processo modelo. Neste caso, a função é chamada "**FUNÇÃO EMBUTIDA SIMPLES**", e quem produz as saídas são operações do próprio processo modelo.

Uma situação em que os eventos não dependam do modelo, mas da ocorrência de um evento externo, esse evento pode ser indicado por um marcador de tempo ou por um registro especial em uma sequência de dados. Uma possível configuração para tal caso seria a ilustrada pela Figura 3.3, onde R representa o marcador de tempo ou o registro especial, M representa um processo modelo, F um processo função e S a saída de um sistema genérico.

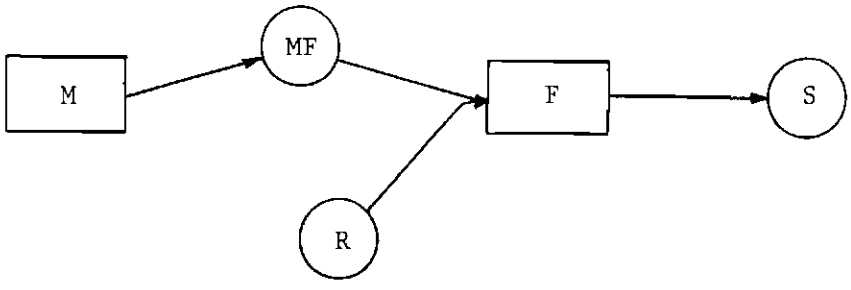


Fig. 3.3 - Exemplo de utilização de um marcador de tempo por um processo função.

Neste caso, a função F funde as duas ocorrências de dados de entrada para produzir a saída S. Caso o registro R contenha um conjunto de variáveis, este fato, gera uma indeterminância na saída da função. Se o número de variáveis for pequeno e o processo F não alterar os seus valores, então a indeterminância de S pode ser suavizada com a seguinte alteração no sistema: R passa a ser parte do processo modelo M constando do seu vetor de estado e F conecta-se à M através do vetor de estados. A indeterminância provocada pela fusão forçada de MF e R passa a ser uma indeterminância própria da conexão por vetor de estados (ver passo de sincronização). A função F, nesta configuração, é chamada **"FUNÇÃO IMPOSTA SIMPLES"**. As variáveis computadas que são inseridas no processo modelo nesta fase, são chamadas "atributos" do processo M. O DES fica como apresentado na Figura 3.4.

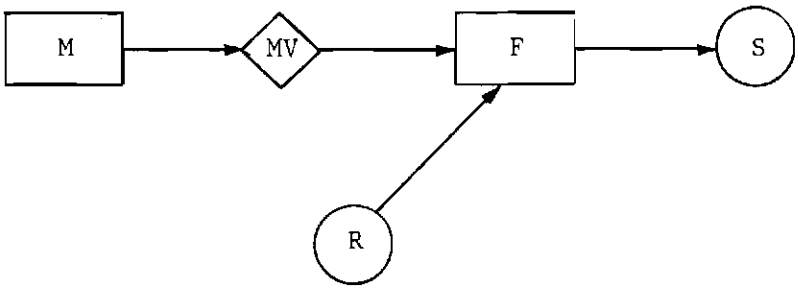


Fig. 3.4 - Exemplo de uma Função Imposta Simples.

Existem funções impostas que não são simples, onde para cada registro de entrada pode ser necessário acessar muitos vetores de estados de entidades diferentes ou acessar vetores de estados em uma certa ordem e buscá-los seletivamente. Em alguns ambientes de implementação pode-se deixar que estas funções sejam manipuladas por software de busca a banco de dados de propósitos gerais.

Um terceiro tipo de função definida no JSD é a **"FUNÇÃO INTERATIVA"**. Este tipo de função interage com o modelo através da leitura do vetor de estados de um dado processo modelo e posterior escrita em uma sequência de dados,a qual entra no mesmo processo modelo por uma fusão forçada com as entradas do mesmo. A Figura 3.5 ilustra uma forma de conexão característica de funções interativas.

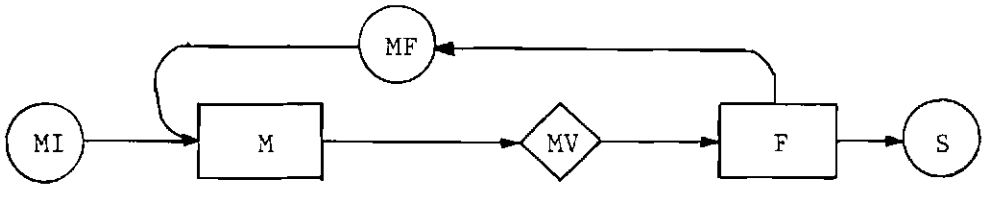


Fig. 3.5 - Exemplo de uma Função Interativa.

Estas formas de conectar os processos funções ao modelo, descritas acima, podem ser combinadas de diferentes maneiras. Uma função pode necessitar mais de um processo função; um processo função pode ser conectado a mais de um processo modelo ou a outros processos funções; um processo função pode ter várias conexões, de tipos diferentes, ligados a um processo modelo; processos funções podem também ser conectados entre si. Cabe determinar quais os processos funções são realmente necessários, como eles devem ser conectados ao DES, bem como, especificar a estrutura de cada processo função e ajustar os processos modelos existentes às novas conexões. Certamente, haverá mais de uma maneira de definir os processos funções e suas conexões, para produzir os requisitos funcionais do sistema.

A fim de auxiliar o analista, responsável pelo desenvolvimento, na escolha de certas alternativas de solução, o JSD estabelece algumas regras:

1) as funções são sempre especificadas como processos com tempo de vida igual a toda a vida do sistema e nunca como procedimentos com tempo de vida curto;

2) um processo modelo só pode ter propósitos funcionais se sua alteração constar apenas da introdução de variáveis e de operações elementares, sem modificar sua estrutura. (Operações elementares seriam, uma escrita, uma leitura, uma operação aritmética, uma atribuição, etc). Caso uma modificação estrutural seja necessária, deve ser especificado um novo processo modelo (considerado um processo do nível 2, cujo nome recebe a extensão -2) conectado ao processo modelo original via uma sequência de dados;

3) para escolher entre uma conexão via vetor de estados ou sequência de dados, o analista pode considerar quem toma a iniciativa de conectar. Se o processo função toma a iniciativa para obter a entrada, a conexão conveniente é por vetor de estado, caso contrário, se a iniciativa reside no processo modelo é mais apropriada uma conexão via sequência de dados.

Em algumas aplicações onde um aspecto mais realístico de controle é exigido, pode ser útil o uso da técnica de backtracking. Esta técnica consiste de sair de situações como uma iteração eterna, onde a condição de término nunca será alcançada. Ela consta de uma declaração "**quit**" especificando o nome de um componente da estrutura e também uma condição de saída. A declaração "quit" pode aparecer em pontos no texto da estrutura, de uma iteração ou de uma seleção de duas partes, onde possa ser verificado se a condição é incorreta.

A instrução "quit" provoca uma saída da iteração se a suposição for incorreta. A sintaxe desta técnica introduz duas novas palavras "posit" e "admit". A parte de "posit" ocorre se a suposição for correta e a parte de "admit" ocorre se a suposição for incorreta. As condições sobre "posit" e "admit" são usadas somente para comentários. A sintaxe do backtracking é apresentada a seguir:

A **posit** (alguma suposição)
...

A **quit** (alguma condição negando a suposição)
...

A **admit** (negação da suposição)
...

A **end**

Muitos detalhes de baixo nível aparecem, principalmente, quando são introduzidas as funções ao sistema. Para Jackson, os detalhes de "baixo nível" são importantes na especificação, porque, sem eles não se pode assegurar que as decisões de **alto nível** estão certas e alto nível é frequentemente um eufemismo para **vago.**

Considerando o processo função em detalhes é possível descobrir se a função pode ser especificada de forma não ambígua, se o conjunto certo de processos funções foi somado ao DES, se os processos funções foram corretamente conectados ao modelo e se o modelo é adequado para satisfazer às funções especificadas. Estas características mostram que o JSD não é um método descendente (top down).

Especificar um sistema com o uso de um modelo automatizado pode, certamente, auxiliar o analista na escolha de alternativas de solução, permitindo-lhe examinar o comportamento do sistema para diferentes configurações. Seria ideal se a especificação pudesse ser executada em computador ao mesmo tempo em que fosse escrita em termos do mundo real do usuário.

3.2.5 - PASSO 5 : SINCRONISMO DO SISTEMA

As questões de tempo afetam as saídas do sistema, no entanto elas não são levantadas no passo das funções. Elas são colocadas em um passo especial entre a especificação e a implementação, por estarem devidamente interrelacionadas a ambos.

Várias considerações de tempo, ou de atraso podem ser observadas no modelo: tempo para as mensagens do mundo real chegarem aos processo do sistema, tempo introduzido quando o processo é implementado, tempo de execução para diferentes sequências de processos realizarem uma função e outros.

Neste passo, devem ser levantados todos os possíveis atrasos do sistema e os tempos desejados para a realização de cada uma das funções (ou saídas) do sistema. Este levantamento deve ser feito pelo analista juntamente com o usuário e/ou comprador. As decisões, que nesta fase podem ser documentadas informalmente, constituem a entrada para o passo de implementação. Elas são condições importantes para a escolha das técnicas de implementação. Restrições de tempo podem significar alteração, até mesmo, na própria especificação.

Em uma especificação JSD a velocidade de execução dos processos não é explícita. Pelo menos em princípio, pode-se imaginar cada processo sendo executado em uma velocidade diferente. Por exemplo, podemos imaginar um processo sendo executado continuamente, um outro sendo executado de hora em hora, outro ainda sendo executado apenas uma vez por semana, e assim por diante, de acordo com as necessidades do sistema.

Essa liberdade de imaginar diferentes velocidades dos processos vai refletir uma liberdade de implementação.

Como foi comentado na seção anterior, uma conexão por vetor de estados tem em si uma indeterminância. Pois, o processo que lê o vetor de estados corre o risco de obter informações já obtidas na última leitura ou perder atualizações feitas entre uma leitura e outra. É claramente notado que deverá haver um ajuste nos tempos dos processos conectados para garantir a consistência das informações trocadas. Outra situação em que há necessidade de ajuste, é quando um processo possui uma fusão forçada com um número muito grande de entradas produzidas por processos diferentes, que podem ser executados em velocidades diferentes, e ainda devendo ser sincronizados via um sinal de marcador de tempo.

Quando é necessário especificar relações de tempo exatas, o JSD providencia uma sincronização adicional aos processos através da introdução de um novo processo no sistema: um processo de sincronização, cuja única função é sincronizar alguns ou todos os processos do sistema. O processo de sincronização é destacado no DES recebendo um traço vertical, a mais, á direita do retângulo, diferenciando-o dos demais processos. É conectado, aos processos que devem ser sincronizados, por meio de sequências de dados.

Com os requisitos de tempo definidos pode-se, finalmente, passar ao passo de implementação, tarefa esta, que pode ser realizada unicamente por pessoas cujo interesse está mais voltado para as técnicas de computação, do que para a aplicação do sistema.

3.2.6 - PASSO 6 : IMPLEMENTAÇÃO

A característica do passo de implementação é tomar a especificação como base e adaptá-la, através de transformações, à máquina disponível. Esta, corresponde à fase de transformação do modelo operacional, apresentado no Capítulo 2.

Neste passo, o analista, responsável pelo desenvolvimento, deve considerar alguns fatores tais como a linguagem de programação, o sistema de gerenciamento e busca de banco de dados, o sistema operacional, enfim, todo o pacote de software disponível e que poderá ser útil na implementação.

Dada a grande variedade de diferentes sistemas que podem ser desenvolvidos e a grande variedade de ambientes de software e hardware disponíveis para o desenvolvimento, o JSD não oferece soluções particulares.

Uma característica da especificação em JSD é a existência de um grande número de processos. Há um processo para cada tipo de entidade e uma instância de processo para cada entidade do mesmo tipo. Isso traz dois problemas. Primeiro, como a maioria dos computadores existentes não comporta um número equivalente de processadores, surge a necessidade de adaptar os processos aos processadores disponíveis. Segundo, quando um número muito grande de processos compartilham um único processador, a memória primária torna-se insuficiente para alocar todos os processos e seus dados.

Pelas razões acima citadas, o passo de implementação concentra-se apenas em resolver os problemas de como compartilhar os recursos entre processos e dados, estabelecendo regras implementáveis de escalonamento e segmentação de processos.

Para mostrar a configuração do sistema que será implementado, o JSD possui um outro diagrama, chamado **Diagrama de Implementação do Sistema - DIS** ('System Implementation Diagram - SID'). Caso os recursos disponíveis fossem suficientes para implementar o próprio DES, o DIS não seria necessário. Porém, como isso quase nunca acontece, foi criado o DIS para mostrar as transformações aplicadas ao sistema para que ele pudesse ser executado. Vários diagramas de implementação podem ser construidos para um mesmo diagrama de especificação, isso quer dizer que há várias formas de implementar um mesmo sistema.

Uma maneira de reduzir o número de processadores necessários para alocar todos os processos é simular vários processadores em um só através de multiprogramação, ou qualquer outra técnica capaz de criar processadores virtuais. Uma outra maneira é transformar a especificação de tal forma a reduzir, somente sob o ponto de vista da implementação, o número de processos sequenciais a serem executados.

O primeiro passo é considerar como os processos podem ser escalonados, isto é, como o tempo de um processador pode ser compartilhado entre os vários processos especificados. Para um ajuste adequado podem ser necessárias algumas trocas no texto dos processos ou mesmo a introdução de um processo escalonador. O escalonador, ao permitir que um processo rode, deve fazer com que a execução continue do ponto, no texto, onde ele tinha parado anteriormente. Além disso, ele pode ser projetado para ler as entradas e escrever as saídas do sistema.

Os processos que compartilham um processador são delimitados com um quadrado desenhado no DES. Sob o ponto de vista da máquina, o escalonador mais os processos subordinados a ele formam um único processo sequencial.

Outra maneira de administrar o escalonamento dos processos é usar a técnica denominada inversão. Ela consta de converter processos conectados por sequência de dados em subrotinas. O processo receptor da sequência de dados é convertido em uma subrotina do processo transmissor da sequência. A técnica de combinar processos por inversão pode ser aplicada a qualquer sistema que satisfaça as condições:

1) o sistema usa apenas conexões por sequência de dados;

2) não existe fusão forçada no sistema;

3) quaisquer dois processos do sistema são conectados por um único caminho.

Para ilustrar a técnica de inversão é apresentado, na Figura 3.6, um exemplo de um sistema imaginário simples.

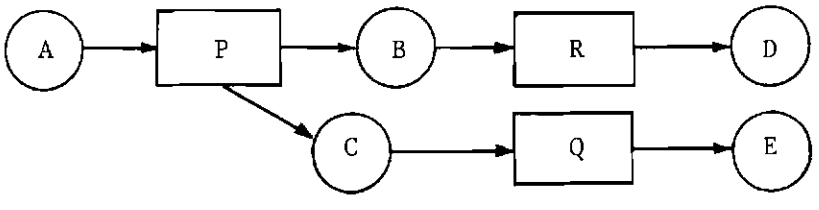


Fig. 3.6 - DES de um sistema imaginário simples.

Como P e Q são ligados por uma sequência de dados, C e o sistema satisfaz as condições citadas acima, então, eles podem ser invertidos com relação a C. Mas P e R também são ligados por uma sequência de dados, B. Logo, R pode ser invertido com relação a B. O esquema de escalonamento deste sistema é:

1) R é executado até que ele seja bloqueado esperando um registro de B, então;

2) P é executado até que ele produza um registro para B, aí P é suspenso e R continua;

3) mas se P, enquanto está executando, produz um registro para C, então, P é suspenso e Q executa até que ele leia o registro de C, aí ele é bloqueado à espera de um novo registro em C.

A Figura 3.7 mostra o DIS para este exemplo.

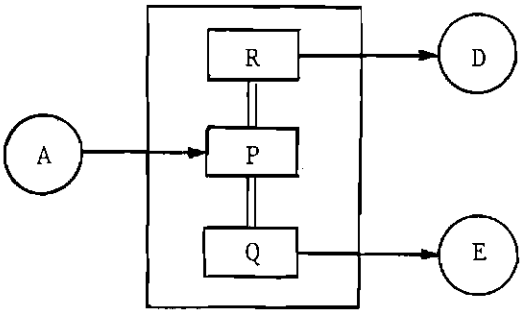


Fig. 3.7 - DIS após a inversão.

As linhas paralelas ligando dois processos no DIS representam uma conexão entre eles através de sequência de dados.

A técnica de inversão pode ser aplicada a um sistema que posssui uma fusão forçada. Imaginemos um processo conectado a outro por mais de um caminho, mas estes caminhos se unem, mais à frente, em uma fusão forçada. O esquema mostrado na Figura 3.8 ilustra este caso típico envolvendo uma fusão forçada.

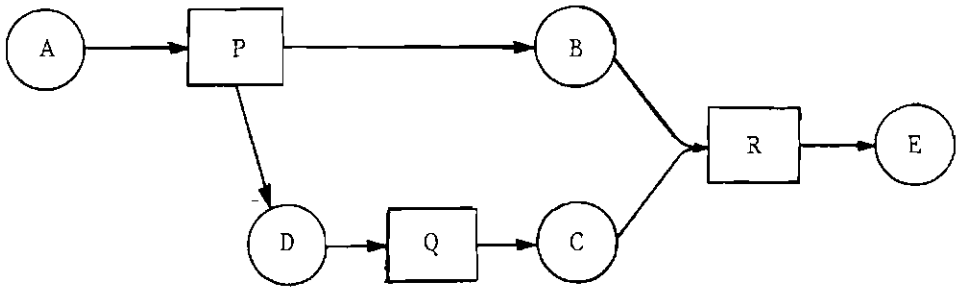


Fig. 3.8 - Exemplo de um sistema envolvendo uma fusão forçada.

Os processos P e Q podem ser invertidos com relação a D, obtendo-se o sistema ilustrado na Figura 3.9.

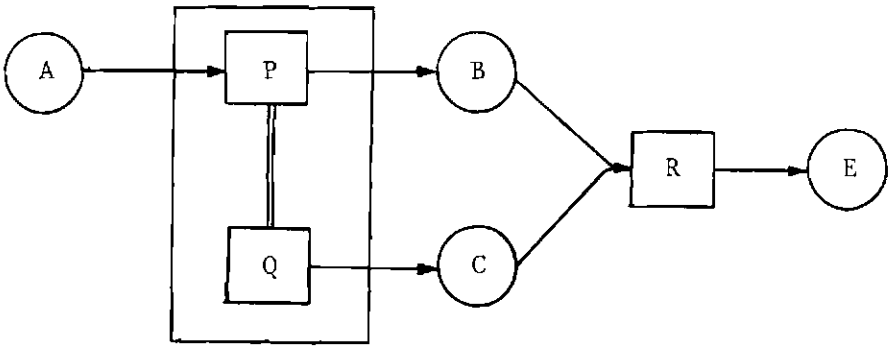


Fig. 3.9 - Sistema com fusão forçada após a inversão de P e Q.

O processo resultante da combinação de P e Q escreverá mensagens nas sequências B e C em alguma ordem. Se esta ordem for aceitável como resultado da fusão forçada em R, então, P pode ser invertido com relação a fusão forçada B&C. R passa a ser, então, uma subrotina dos dois processos, P e Q. O DIS resultante das transformações do sistema é ilustrado na Figura 3.10.

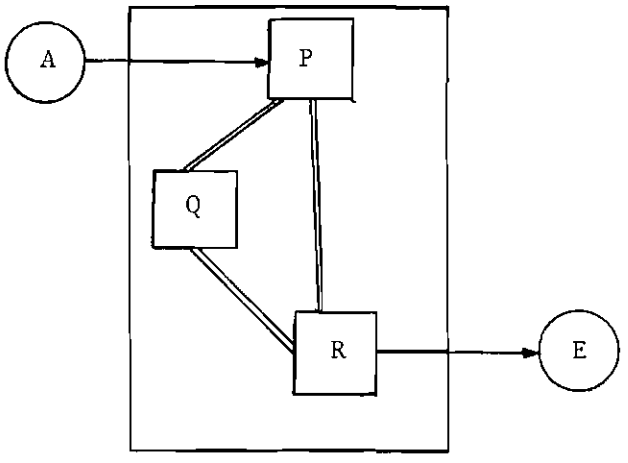


Fig. 3.10 - DIS de um sistema invertido com relação a uma fusão forçada.

A notação usada para as conexões com o processo R é uma extensão da notação de inversão por sequência de dados. Como vimos, uma fusão forçada pode ser implementada usando a técnica de inversão, onde o processo que recebe a fusão forçada transforma-se em uma subrotina chamada por todos os processos que produzem entradas para a fusão forçada.

Quando um processo é conectado a outro por duas ou mais sequências de dados, este processo também pode ser invertido. A inversão se dá sobre todas as sequências de dados e a nova conexão é chamada conexão de canal, representada no DIS por três ou mais linhas paralelas que ligam os processos invertidos. O exemplo a seguir, pode esclarecer melhor a situação. Dado o sistema ilustrado na Figura 3.11 deseja-se implementá-lo com um processo escalonador. Ao inverter os três processos com relação as sequências de dados que os ligam ao escalonador tem-se o DIS da Figura 3.12. A barra tripla que liga o processo Q ao ESCALONADOR indica uma conexão de canal, ou seja, indica que o processo foi invertido com relação às duas sequências de dados, no caso, B e C.

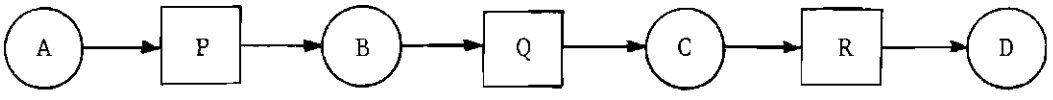


Fig. 3.11 - Exemplo de um sistema com um processo conectado a duas sequências de dados.

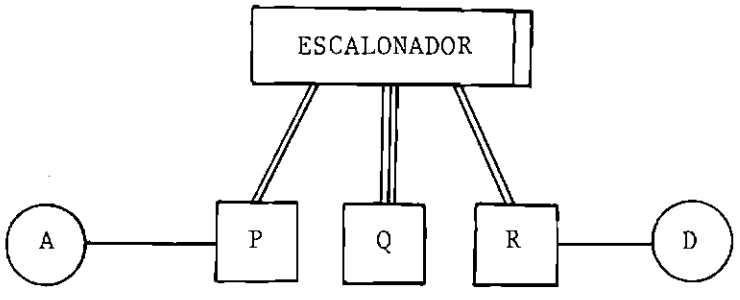


Fig. 3.12 - DIS com uma conexão de canal.

Algumas aplicações exigem que processos guardem informações recebidas para serem processadas depois de um tempo decorrido. Neste caso, buffers são introduzidos ao sistema. Esta solução é sempre evitada pelo JSD porém quando necessário os buffers ficam sob o controle do escalonador.

Quando é preciso reduzir área de memória ocupada por vetores de estados, uma solução é armazenar os vetores de estados de todos os processos em arquivos pertencentes à memória secundária (discos e fitas), e atribuir ao escalonador a função de buscar os registros dos vetores de estados. Pode-se usar aqui, o sistema de busca do banco de dados existente, caso houver, ou inserir em cada processo uma operação que busque (loadsv) e/ou armazene (storesv) o vetor de estados.

Outra solução é separar o vetor de estados do texto do processo e manter uma cópia do vetor de estados para cada instância de processo e somente uma cópia do texto executável para todas as instâncias.

Situações em que o texto de programa do processo não cabe na memória principal, o JSD padronizou uma outra transformação: o desmembramento do processo. Um desmembramento do processo é sempre acompanhado de um desmembramento das sequências de dados conectadas a ele. Há dois modos de se desmembrar um processo:

1) retirar partes do texto do processo que serão executadas somente quando uma determinada condição ocorrer. Este caso é conhecido como desmembramento condicional. A Figura 3.13 ilustra um desmembramento condicional onde o processo Q foi dividido e suas partes, Qa e Qb, são executadas em condições distintas. As partes desmembradas são representadas no DIS em caixas separadas. O nome do processo recebe uma letra minúscula como extensão para distinguir suas partes;

2) separar o processo de tal forma que parte dele seja executada em uma hora ou dia produzindo um resultado intermediário que servirá como entrada para a parte executada em outra hora ou dia. Esta técnica é conhecida como desmembramento temporal. Ela é especial para implementação de programas (em batch) que podem ser executados periodicamente. Desta forma faz-se necessário uma segmentação de programa e de área de dados. A figura 3.14 mostra um desmembramento temporal onde as sequências de dados, A e B, também foram desmembradas. A parte das sequências ligadas à parte do processo com extensão b, recebem o mesmo sufixo em seus nomes. A notação usada no DIS para representar tal tipo de desmembramento é mostrada na Figura 3.14.

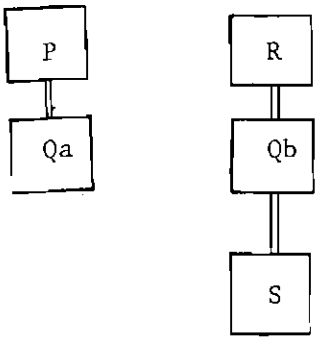


Fig. 3.13 - DIS mostrando um desmembramento condicional.

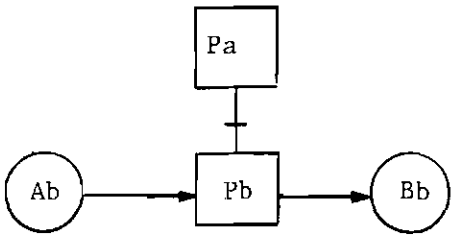


Fig. 3.14 - DIS mostrando um desmembramento temporal.

O Capítulo 4 descreve uma ferramenta de software que dá suporte à elaboração de sistemas construidos segundo o método JSD cobrindo os cinco primeiros passos do método. O Capítulo 5 traz as características do Executor-JSD, uma ferramenta que cobre o sexto passo do método, isto é, executa uma especificação JSD. O Capítulo 6 mostra um exemplo de um sistema imaginário de uma competição promovida pelo jornal Daily Racket que é descrito no livro "System Development" por Michael Jackson. Este exemplo foi implementado e pode ser executado pelo Executor-JSD.

CAPÍTULO 4

O JSD-TOOL

O JSD-tool, ou simplesmente tool, é uma ferramenta de software definida para ajudar analistas e usuários a escreverem e analisarem uma especificação produzida segundo o método JSD. Ela conta com uma linguagem que cobre os cinco primeiros passos do método (ações e entidades, estrutura da entidade, modelo inicial, função e sincronismo do sistema). Seguindo a sintaxe desta linguagem o usuário é capaz de escrever sua especificação em JSD. Utilizando os comandos próprios do JSD-tool ele pode, criar um arquivo contendo a especificação, corrigir os erros de sintaxe, resolver os problemas de consistência e completude da especificação, guardar um histórico das decisões tomadas no decorrer do desenvolvimento, desde que salve todos os arquivos e relatórios gerados pelo tool, etc.

Uma especificação criada com o auxílio do JSD-tool, pode ser construída e verificada passo a passo, como sugere a metodologia do Jackson. A verificação é feita através de cheques automáticos de consistência. Diferentes verificações são efetuadas em cada passo e mostradas ao usuário através de relatórios.

O JSD-tool é inicializado quando o usuário digita **jsd**. Ao ser inicializado, o sistema abre uma sessão de trabalho, a qual é caracterizada por ter dois estados ou modos de operação diferentes. São eles: modo de especificação e modo de análise. Cada modo tem uma forma específica de comunicação com o usuário. As Seções 4.1 e 4.2 descrevem as interfaces com o usuário, bem como, as características particulares destes dois modos de operação do tool (VELASCO, 1986b).

4.1 - MODO DE ESPECIFICAÇÃO

Ao entrar no JSD-tool o usuário cai automaticamente no modo de especificação, pois este é o modo padrão de trabalho no tool. Neste momento, tem início uma sessão no modo de especificação: o sistema mostra o sinal "JSD:" no terminal de vídeo e fica esperando um comando do usuário.

A comunicação do JSD-tool com o usuário, aqui, se dá através de comandos, os quais são divididos em dois tipos:

1) comandos de especificação, também chamado declarações de especificação;

2) comandos de controle.

4.1.1 - COMANDOS DE ESPECIFICAÇÃO

Os comandos de especificação são aqueles que vão formar uma especificação. Eles são divididos de acordo com os passos do método JSD, no qual eles devem ser usados. Os comandos que pertencem ao passo de Ações e Entidades devem ser usados quando o desenvolvimento encontra-se no primeiro passo do método, e isso vale também para os demais comandos: eles devem ser usados dentro do passo ao qual pertencem. Desta forma, passo a passo, a especificação vai sendo criada. Como o JSD-tool não cobre o passo seis, não há comandos para a fase de implementação.

O conjunto de comandos de um passo posterior sempre engloba os comandos dos passos anteriores. Cada passo seguinte introduz novos comandos ou mantém os comandos do passo anterior acrescidos de uma expressão com maior poder explicativo, como pode ser visto na descrição apresentada nas seções posteriores. Cabe observar que neste modo de operação, apenas a análise sintática dos comandos pode ser realizada.

4.1.1.1 - PASSO DE AÇÕES E ENTIDADES

Os comandos para especificação do passo de ações e entidades são:

1) **sysname** - define ou troca o nome que identifica o sistema que está sendo especificado;

2) **entity** - define e adiciona uma entidade à especificação junto com suas ações associadas;

3) **actio**n - define e adiciona uma ação à especificação junto com seus atributos.

4.1.1.2 - PASSO DA ESTRUTURA DA ENTIDADE

Os comandos para especificação do passo da estrutura da entidade englobam todos os anteriores mais os comandos itemizados abaixo, os quais servem para definir a estrutura da entidade determinando seu nome e sua composição. Estes comandos são chamados comandos de estrutura. São eles:

1) **sequence** - define um nome de sequência como uma sequência de nomes de itens. Onde um item pode ser o nome de uma ação ou o nome de uma estrutura (isto é, um nome de sequência, um nome de seleção, ou ainda um nome de iteração);

2) **selection** - define um nome de seleção como uma seleção de nomes de itens, com uma cláusula vazia opcional no final do comando;

3) **iteration** - define um nome de iteração como um nome de item;

4.1.1.3 - PASSO DO MODELO INICIAL

Os comandos para especificação do passo do modelo inicial englobam todos os comandos do passo de ações e entidades, os comandos de estrutura, mais alguns comandos que definem um grafo direcionado rotulado. Este grafo representa o Diagrama de Especificação do Sistema (DES) onde, os nós representam processos e os arcos representam a comunicação entre os processos (isto é, as conexões). O grafo é também chamado modelo de trabalho, nele as conexões são definidas pelos comandos de conexão (external stream, external state, stream e state) e pelos comandos de fusão forçada (rough merge). Os nós do grafo são definidos através de um nome de processo. A definição de uma conexão possui três componentes: um tipo (vetor de estados ou sequência de dados), um nome e uma indicação da multiplicidade (um-para-um, um-para-muitos, muitos-para-um, muitos-para-muitos). Os processos são definidos pelos comandos de estrutura (os mesmos definidos no passo da Estrutura da Entidade) acrescidos de itens que permitem a descrição da comunicação entre os processos. Os comandos do passo do modelo inicial são:

1) **sequence** - define um nome de sequência como uma sequência de itens. Onde um item, neste passo, além de poder ser o nome de uma ação ou um outro nome de uma estrutura pode ser uma declaração de entrada saída: **read** - comando para ler uma certa sequência de dados, **write** - comando para escrever um nome de ação em uma certa sequência de dados, **getsv** - comando para obter um certo vetor de estados;

2) **selection** - define um nome de seleção como uma seleção de itens, com uma cláusula vazia opcional no final do comando. Condições (texto livre) devem ser atribuídas a cada parte da seleção a fim de especificar qual parte deve ser escolhida;

3) **iteration** - define um nome de iteração como uma lista de itens. Uma condição (texto livre) deve ser atribuída ao comando a fim de especificar quantas vezes ele deve ser iterado;

4) **external stream** - define uma conexão por sequência de dados entre o mundo real e um processo. Este último fica definido implicitamente;

5) **external state** - define uma conexão por vetor de estado entre o mundo real e um processo. Este último fica definido implicitamente;

6) **stream** - define uma sequência de dados que conecta dois processos, e estabelece a relação de multiplicidade da sequência de dados;

7) **state** - define um vetor de estados que conecta dois processos, e estabelece a relação de multiplicidade dos vetores de estados;

8) **rough merge** - indica quais sequências de dados compõem uma fusão forçada.

4.1.1.4 - PASSO DE FUNÇÃO

Os comandos para especificação do passo de função podem incluir os comandos do passo de ações e entidades, devem incluir os comandos que definem o grafo direcionado rotulado, ou seja, o DES, e os comandos de estrutura que vão compor a estrutura dos processos. Novos comandos são definidos neste passo a fim de permitir a utilização dos novos artifícios introduzidos aos processos e as conexões no passo de Função do método JSD. Os novos tipos de conexões, além daquelas definidas no passo anterior são: conexão por sequência de dados para entradas do mundo real ao sistema, por sequência de dados para saídas do sistema para o mundo real, por sequência de um marcador de tempo, definidas respectivamente pelos comandos, enquiry, report e time. Na definição da estrutura de processos são introduzidos outros comandos a fim de permitir que uma seleção de duas partes ou uma iteração possam expressar a técnica de backtracking. Os comandos disponíveis no passo de função são:

1) **sequence** - define um nome de sequência como uma sequência de itens. Onde um item, neste passo, além de poder ser o nome de uma ação, ou o nome de uma estrutura, ou um dos comandos: **read**, **write**, ou **getsv**, pode ser também um texto; e o comando write além de escrever um nome de ação pode escrever um texto em uma certa sequência de dados;

2) **selection** - define um nome de seleção como uma seleção de itens, com uma cláusula vazia opcional no final do comando. Condições booleanas devem ser atribuídas a cada item;

3) **iteration** - define um nome de iteração como uma lista de itens, seguida por zero ou mais instruções "**quit**".(quit é seguido por uma condição booleana mais uma lista de itens);

4) **posit** - define um nome de posit como uma lista composta de uma condição booleana mais um conjunto de itens encerrada com uma cláusula **admit**;

5) **enquiry** - define uma sequência de dados que gera uma entrada do mundo real a um processo interno. Ele também define um novo processo, implicitamente.

6) **time** - define uma sequência de dados que gera uma entrada de tempo a um processo interno. Ele também define um novo processo, implicitamente.

7) **report** - define uma sequência de dados gerada por um processo, sequência esta que liga o processo interno ao mundo real. Ele também define um novo processo, implicitamente.

Aqui encerra a lista dos comandos de especificação, a sintaxe dos mesmos é encontrada no manual de referência (VELASCO, 1986a).

O passo de sincronização do sistema não possui comandos explícitos. A sincronização deve ser feita manualmente; nesta fase, o texto da estrutura e o próprio diagrama de especificação do sistema (DES) podem sofrer alterações para se obter a melhor temporização das saídas desejadas do sistema.

4.1.2 - COMANDOS DE CONTROLE

Os comandos de controle são usados para ler ou salvar uma especificação, trocar o modo de operação do JSD-tool, deixar o sistema, enfim, direcionar o uso da ferramenta. O JSD-tool dispõe dos seguintes comandos de controle:

1) analysis - troca o modo de operação para o modo de análise;

2) help - imprime a sintaxe dos comandos;

3) quit - encerra uma sessão de especificação e volta ao sistema operacional.

4) read - lê o arquivo de especificação, o qual deve conter apenas comandos de especificação;

5) reset - apaga a especificação e retorna ao estado inicial da sessão de trabalho;

6) system - permite ao usuário executar comandos do sistema operacional sem sair do tool;

7) write - escreve a especificação em um arquivo designado pelo usuário.

A sintaxe destes comandos pode ser encontrada no manual de referência (VELASCO, 1986a). Na maioria deles o nome do comando é seguido de, um texto quando necessário e, ponto e vírgula para encerrar. O comando help auxilia o usuário iniciante mostrando, em tempo de execução, a sintaxe do comando solicitado.

4.2 - MODO DE ANÁLISE

A passagem para o modo de análise é realizada pelo comando de controle "analysis". Quando o usuário passa para este modo (o que corresponde a entrar em uma sessão de análise), a comunicação, dele, com o JSD-tool deixa de ser através de comandos e passa a ser através de menus.

Neste modo de operação, a especificação pode ser analisada com o objetivo de checar sua completude e consistência. A verificação de erros de sintaxe não é feita aqui. Neste ponto, a especificação deve estar sintaticamente correta, caso contrário não se pode garantir a qualidade da análise.

No início de uma sessão de análise o "menu principal" é apresentado ao usuário. Este menu permite a escolha de um dos passos do desenvolvimento ou a volta ao modo de especificação. Escolhendo um dos passos, um outro tipo de menu, chamado "menu do passo", é mostrado no vídeo. Diante do novo menu, o usuário, pode escolher um ou mais tipos de relatórios, consequentemente ele escolhe um tipo de verificação que o sistema pode fazer sobre a especificação.

Para cada passo do método JSD, três tipos de relatórios podem ser gerados:

1) relatório de consistência e completude: neste tipo de relatório é criada uma lista que relaciona os pontos da especificação onde as restrições de consistência e completeza não foram obedecidas. A observação de tais pontos irão garantir a corregibilidade da especificação;

2) relatório de referência e sumário: este tipo de relatório mostra o conteúdo da especificação sob vários formatos e pontos de vista diferentes. Eles podem servir como uma base para a documentação do sistema;

3) relatório de análise: este tipo revela vários aspectos diferentes da especificação, como por exemplo, dado um processo P, a análise pode relatar todos os processos conectados a P.

Para maior compreensão da importância dos vários relatórios, bem como o conteúdo exato dos mesmos, as Seções 4.2.1, 4.2.2 e 4.2.3 trazem uma descrição de como é feita a análise sobre a especificação para obtenção dos vários relatórios, e o conteúdo de cada um dos tipos de relatórios oferecidos ao usuário.

4.2.1 - ANÁLISE DE CONSISTÊNCIA E COMPLETUDE

Algumas restrições de consistência e completude devem ser satisfeitas por uma especificação escrita com o auxílio do JSD-tool. Tais restrições atingem somente os quatros primeiros passos do método JSD. Elas não dizem respeito às restrições impostas pela própria sintaxe da linguagem, mas sim às restrições puramente internas à especificação.

O usuário ao escolher um número de 0 a 5 no "menu principal", obtém o "menu do passo" correspondente ao passo do método JSD que ele deseja analisar. Escolhido um dos passos, ele pode solicitar o relatório de consistência e completude digitando a opção "Problems" no menu do passo que lhe foi apresentado. A análise feita sobre a especificação para o obtenção do relatório "Problems" é feita com base nas restrições impostas pelo JSD-tool em cada passo. As Seções seguintes descrevem, para cada um dos passos de um a quatro, as restrições impostas à especificação e o conteúdo do relatório "Problems".

4.2.1.1 - PASSO DE AÇÕES E ENTIDADES

As restrições de consistência e completude para este passo são:

1) pelo menos uma entidade deve ser definida;

2) pelo menos uma ação deve ser definida;

3) toda ação listada em um comando entity deve ser definida em um comando action;

4) toda ação definida em um comando action deve pertencer a lista de ações de algum comando entity.

O relatório de problemas deste passo contém:

1) uma relação de todas as entidades que possuem ações indefinidas, mostrando o nome da entidade seguido dos nomes das ações que pertencem a ela mas não foram definidas;

2) uma relação de todas as ações que foram definidas mas não foram associadas a alguma entidade.

4.2.1.2 - PASSO DA ESTRUTURA DA ENTIDADE

As restrições de consistência e completude para este passo são:

1) não pode haver estrutura que dependa dela mesma, isto é, não são permitidas definições circulares;

2) não são permitidas entidades indefinidas, ou seja, toda entidade deve estar definida em algum comando de estrutura;

3) nenhuma ação pode ser definida em um comando de estrutura, pois toda ação deve ser uma folha de alguma estrutura;

4) todo nó folha contém exclusivamente uma ação;

5) não pode haver definições inconsistentes de entidades, isto é, o conjunto de ações de uma entidade deve ser o mesmo conjunto de ações do qual esta entidade depende.

O relatório problems do passo de Estrutura da Entidade contém uma lista de:

1) todas as definições circulares;

2) todas as entidades indefinidas;

3) todas as ações definidas como estrutura;

4) todos os nós folhas que não contêm uma ação;

5) todas as definições de entidade inconsistentes.

4.2.1.3 - PASSO DO MODELO INICIAL

As restrições de consistência e completude para o passo do modelo inicial podem ser divididas em três classes:

1) boa-formação do modelo de trabalho ou modelo inicial;

2) consistência e completude do modelo de trabalho com relação a descrição das entidades e ações;

3) consistência e completude do modelo de trabalho com relação às estruturas de processo especificadas;

As restrições para a primeira classe de análise são:

1) não pode haver ciclos no modelo inicial, isto é, ele deve ser um grafo acíclico;

2) não pode haver processos não alcançáveis, isto é, todo processo modelo deve ser alcançável por um processo do nível 0;

3) não pode haver múltiplas conexões com o mundo real (externo), isto é, nenhum processo modelo pode ser o destino de mais de uma conexão externa;

4) não pode haver fusão forçada inválida, isto é, a lista de fusão forçada deve ser disjunta e ser composta de conexões por sequência de dados que tenham todas o mesmo processo destino;

5) a definição de uma estrutura não pode ser circular, isto é, nenhuma estrutura pode depender dela própria.

As restrições da classe que analisa o modelo inicial versus a descrição das ações e entidades são:

1) não pode haver processos que não sejam entidades, isto é, todo nome de processo deve ser também um nome de entidade;

2) não podem desaparecer entidades, isto é, todo nome de entidade deve ser também um nome de processo;

3) não pode haver ações impossíveis de serem geradas, isto é, toda ação deve pertencer a lista de ações de pelo menos uma entidade, tal que o processo correspondente a esta entidade tenha uma conexão externa como entrada;

4) não pode haver ações não consumíveis, isto é, se uma ação ocorre em uma lista de ações de uma entidade, então, o processo que corresponde a esta entidade deve ser ligado a um processo capaz de gerar esta ação;

5) não é permitida a existência de conexões entre processos que não compartilham ações.

As restrições da classe que analisa o modelo inicial com relação a definição das estruturas de processo são:

1) não pode haver desaparecinmento de estruturas de processo, isto é, todo processo modelo deve ser definido por uma estrutura de processo;

2) não pode haver desaparecimento de declarações **read**, isto é, toda sequência de dados do modelo deve ser lida pelo processo destino da conexão;

3) não pode haver declarações **read** incorretas, ou seja, se "**read** A1" ou "**read** A1&A2&..." ocorrem na definição estrutural de um processo P, então o seguinte é verdadeiro:

   a) o proceso P é o processo destino das conexões por sequência de dados A1, A2, ... ;

   b) (no caso de A1&A2&...) as sequências de dados A1, A2,... pertencem a uma fusão forçada.

4) não pode haver desaparecimento de declarações **write**, isto é, para toda conexão por sequência de dados A, do modelo inicial no qual P é o processo fonte, deve existir um item da forma "**write** nome-ação: A" na definição estrutural de P;

5) não pode haver declarações **write** incorretas, ou seja, se uma declaração "**write** nome-ação: A" ou "**write** nome-ação: A(identificador)" ocorrer na definição estrutural de um processo P, então é verdade que:

   a) o proceso P é o processo fonte da conexão por sequência de dados A;

   b) a ação nome-ação pertence a P e ao processo destino da conexão por sequência de dados A;

   c) (no caso de "**write** A") nenhuma identificação do processo desapareceu. A é uma conexão do tipo um-para-um ou do tipo muitos-para-um;

   d) (no caso de "**write** A(identificador)") a identificação do processo é necessária pois, A deve ser uma conexão do tipo um-para-muitos ou muitos-para-muitos;

6) não pode haver desaparecimento de declarações **getsv**, isto é, para toda conexão por vetor de estados A de um modelo inicial no qual P é o processo destino, deve existir um item da forma "**getsv** A" ou "**getsv** A(identificador)" na definição estrutural de P;

7) não pode haver declarações "**getsv**" incorretas, ou seja, se uma declaração "**getsv** A" ou "**getsv** A(identificador)" pertence a definição estrutural de um processo P, então é verdade que:

   a) o processo P é o processo destino da conexão por vetor de estados A;

b) (no caso de "**getsv** A") nenhuma identificação do processo desapareceu, pois A deve ser uma conexão do tipo um-para-um ou muitos-para-um;

c) (no caso de "**getsv** A(identificador)") a identificação do processo é necessária pois, A deve ser uma conexão do tipo um-para-muitos ou muitos-para-muitos.

O relatório de problemas deste passo cobre a análise de todas as classes de restrições relacionadas acima. Este relatório também está dividido em classes de restrições onde, cada classe contém uma lista de itens analisados e a cada item analisado, segue uma lista de pontos que tornam a especificação inconsistente e incompleta. O relatório de problemas do passo do modelo inicial contém:

1) na análise da boa-formação do modelo inicial, o relatório comtém uma lista de:

   a) todos os ciclos do modelo inicial;

   b) todos os processos não alcançáveis;

   c) todas as conexões, entre o mundo real e modelo, que são múltiplas;

   d) todas as fusões forçadas inválidas;

   e) todas as estruturas de seleção e iteração que não contêm uma condição especificada;

   f) todas as estruturas com definições circulares;

2) na análise do modelo inicial versus as descrições das ações e entidades, o relatório contém uma lista de:

a) todos os processos que não possuem uma entidade correspondente;

b) todas as entidades desaparecidas;

c) todas as ações que não podem ser geradas;

d) todas as ações não consumidas por algum processo;

e) todos os processos que são ligados diretamente mas não possuem ações comuns.

3) na análise do modelo inicial versus as estruturas de processos, o relatório contém uma lista de:

a) todas as estruturas de processos que desapareceram, isto é, que não foram definidas;

b) todas as declarações de **read** que desapareceram, para isso verifica se todas as sequências de dados são lidas;

c) todas as declarações de **read** que estão incorretas ou inconsistentes;

d) todas as declarações de **write** que desapareceram, para isso verifica se há comandos de escrita para todas as sequências de dados definidas no sistema;

e) todas as declarações de **write** que estão incorretas ou inconsistentes;

f) todas as declarações de **getsv** que desapareceram, para isso verifica se todos os vetores de estados são acessados via um comando getsv;

g) todas as declarações de **getsv** que estão incorretas ou inconsistentes;

4.2.1.4 - PASSO DE FUNÇÃO

As restrições de consistência e completude impostas sobre este passo são divididas em duas categorias:

1) boa-formação do Diagrama de Especificação do Sistema (DES);

2) consistência e completude do DES com relação as estruturas de processo;

As restrições com relação a boa-formação do DES são:

1) não pode haver processos sem uso, isto é, o mundo externo deve ser atingido a partir de qualquer processo interno. Um processo sem uso pode ser extraído da especificação sem alterar as saídas do sistema;

2) não pode haver processos não alcançáveis. (Como no passo do Modelo Inicial);

3) não pode haver fusão forçada inválida. (Como no passo do Modelo Inicial);

4) não pode haver comandos de iteração e seleção sem expressão condicional. Toda iteração e seleção deve ter condições booleanas que sinalizem o fim da iteração ou a alternativa a ser selecionada;

5) não pode haver definição de estruturas circulares, isto é, a linguagem não permite construções recursivas.

As restrições da categoria que analisa o DES versus as estruturas de processo são:

1) não pode haver desaparecimento de estruturas de processos, isto é, todo processo no DES deve ser definido por uma estrutura de comandos;

2) não pode haver desaparecimento de declarações **read**, isto é, toda sequência de dados no DES deve ser lida pelo processo destino da conexão. (Como no passo do modelo Inicial);

3) não pode haver declarações **read** incorretas. (Como no passo do modelo Inicial);

4) não pode haver desaparecimento de declarações **write**. (Como no passo do modelo Inicial);

5) não pode haver declarações **write** incorretas, ou seja, se uma declaração "**write** nome-registro: A" ou "**write** nome-registro: A(identificador)" ocorrer na definição estrutural de um processo P, então é verdade que:

   a) o processo P é o processo fonte da conexão por sequência de dados A;

   b) (no caso de "**write** nome-registro: A") a identificação do processo não desapareceu, pois A é uma conexão do tipo um-para-um ou muitos-para-um;

   c) (no caso de "**write** nome-registo: A(identificador)") a identificação do processo é necessária pois, A deve ser uma conexão do tipo um-para-muitos ou muitos-para-muitos;

6) não pode haver desaparecimento de declarações **getsv**. (Como no passo do Modelo Inicial);

7) não pode haver declarações "**getsv**" incorretas. (Como no passo do Modelo Inicial).

O relatório que cobre inconsistência e incompletude para o passo de função, chamado relatório de problemas, contém:

1) uma análise da boa-formação do DES contendo uma lista de:

   a) todos os processos sem uso;

   b) todos os processos não alcançáveis;

   c) todas as fusões forçadas inválidas;

   d) todas as estruturas de seleção e iteração que não contêm uma expressão condicional especificada;

   e) todas as estruturas com definições circulares;

2) uma análise do DES comparando-o com a definição das estruturas de processo, contendo uma lista de:

   a) todas as estruturas de processo que desapareceram, isto é, que não foram definidas;

   b) todas as declarações de **read** que desapareceram;

   c) todas as declarações de **read** que estão incorretas ou inconsistentes;

   d) todas as declarações de **write** que desapareceram;

e) todas as declarações de **write** que estão incorretas ou inconsistentes;

f) todas as declarações de **getsv** que desapareceram;

g) todas as declarações de **getsv** que estão incorretas ou inconsistentes;

4.2.2 - ANÁLISE DE REFERÊNCIA E SUMÁRIO

As informações contidas nos relatórios do tipo análise e sumário permitem que o usuário tenha apenas uma parte da especificação em um dado momento. Eles podem ser produzidos mesmo que a especificação viole as restrições de consistência e completude. Porém, se isso acontece a interpretação do conteúdo destes relatórios pode não ser significativa. Para cada passo, do primeiro ao quarto, do método JSD é possível obter-se diferentes relatórios.

4.2.2.1 - PASSO DE AÇÕES E ENTIDADES

Os relatórios que podem ser produzidos neste passo são mostrados ao usuário através de uma lista de opções que fazem parte do "menu do passo 1". Os vários tipos de relatórios de referência e sumário produzidos no passo um são:

1) Entidades: contém uma lista de todas as entidades definidas no sistema;

2) Entidades e Ações: mostra o conjunto de ações definidas em cada entidade;

3) Ações: contém uma lista de todas as ações definidas no sistema;

4) Ações e Descrições: mostra o texto que descreve o significado de cada uma das ações;

5) Ações e Entidades: mostra a relação entre as entidades e as ações já definidas;

6) Ações e Atributos: contém a relação de todos os atributos pertencentes a cada entidade;

7) Atributos: contém uma lista de todos os atributos definidos no sistema;

8) Atributos e Ações: para cada atributo, mostra as ações com as quais o atributo está associado.

Cabe lembrar que, há uma opção no menu do passo que produz um relatório contendo todos os itens citados acima.

4.2.2.2 - PASSO DA ESTRUTURA DA ENTIDADE

Os relatórios de referência e sumário que podem ser produzidos neste passo são mostrados ao usuário através de uma lista de opções pertencentes ao "menu do passo 2". Assim,o usuário pode escolher os tipos de relatórios que desejar. Neste passo as opções são:

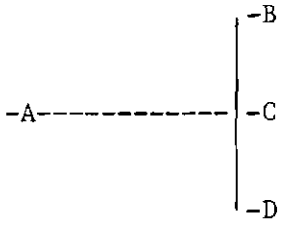
1) Dependência da Estrutura: para cada estrutura definida no sistema, mostra as outras estruturas das quais ela depende;

2) Definição da Estrutura: para cada estrutura definida, é mostrado o conjunto de ações do qual a estrutura depende;

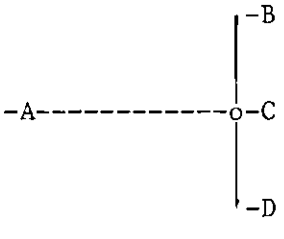
3) Texto da Estrutura da Entidade: para cada entidade, as definições das estruturas sobre a qual a entidade depende são combinadas para dar a estrutura textual da entidade;

4) Diagrama da Estrutura da Entidade: para cada entidade, é mostrado a árvore de estrutura da entidade. As convenções para a construção são:

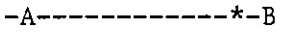
a) uma sequência **seq** A: B, C, D;



b) uma seleção **sel** A: B, C, D;



c) uma iteração **iter** A: B;



4.2.2.3 - PASSO DO MODELO INICIAL

Os relatórios de referência e sumário que podem ser produzidos neste passo são apresentados ao usuário através do "menu do passo 3". Frente a este menu o usuário pode escolher um dos tipos de relatórios desta categoria (referência e sumário). Neste passo as opções são:

1) Processos: mostra o nome de todos os processos definidos no sistema;

2) Processos e Conexões: para cada processo, mostra todas as conexões que são lidas e todas as que são escritas pelo processo;

3) Conexões: dá o tipo (sequência de dados externa, vetor de estados externo, sequência de dados, vetor de estados), os processos fonte, os processos destino e o rótulo de multiplicidade de cada conexão definida;

4) Fusão forçada: para cada processo, mostra as sequências de dados que são lidas e fundidas, e também de quais processos estas sequências se originam;

5) Sequências de Dados que Não Pertencem a Fusão Forçada: para cada processo, mostra as sequências de dados que são lidas mas não pertencem a alguma fusão forçada;

6) Texto de Estrutura de Processo: mostra a estrutura textual de todos os processos definidos;

7) Diagrama de Estrutura do Processo: mostra o diagrama de estrutura de todos os processos definidos;

8) Diagrama de Especificação do Sistema: mostra o grafo direcionado que define o modelo inicial. Uma descrição da notação usada pode ser encontrada no manual de referência (VELASCO, 1986a).

4.2.2.4 - PASSO DE FUNÇÃO

Os relatórios de referência e sumário que podem ser produzidos no passo de função são:

1) Processos: mostra uma lista de todos os processos definidos no sistema;

2) Processos e Conexões: para cada processo, mostra as conexões que são lidas e escritas por ele;

3) Conexões: para cada conexão, mostra seu tipo (sequência de dados externa, vetor de estados externo, sequência de dados, vetor de estados, sequência de dados só de entrada (enquiry), sequência de dados só de saída (report), sequência de dados de tempo), os processos fontes e os processos destinos;

4) Sequência de Dados de Entrada e Saída: mostra somente as conexões que se ligam ao mundo real, agrupadas em sequências de entrada e de sequências de saída;

5) Fusão forçada: Como no passo do Modelo Inicial;

6) Texto da Estrutura de Processo: Como no passo do Modelo Inicial;

7) Diagrama da Estrutura do Processo: mostra a árvore de estrutura de cada processo definido;

8) Diagrama da Especificação do Sistema: mostra o grafo direcionado (DES).

Estes relatórios são escolhidos pelo usuário no "menu do passo 4". A opção para obter todos os relatórios em um também pode ser encontrada neste menu.

4.2.3 - ANÁLISE DE OUTROS ASPECTOS

Outros tipos de análise, que procuram mostrar as consequências da especificação podem ser feitas, sobre os cinco primeiros passos do método JSD, pelo JSD-tool. Estas análises podem ser feitas sobre uma especificação ainda incompleta ou com alguns erros. É óbvio que se a especificação encontra-se em um desses estados o resultado da análise não pode ser significativo, entretanto, ela será realizada se assim o usuário o desejar. Diferentes análises são oferecidas a cada passo do método. Cada tipo de análise gera um tipo de relatório. Os relatórios que mostram as consequências da especificação são chamados relatórios de análises e são apresentados nas seções seguintes, separados nos vários passos do JSD.

4.2.3.1 - PASSO DE AÇÕES E ENTIDADES

Os relatórios de análise do primeiro passo podem fornecer informações úteis para a construção do modelo de trabalho (ou Modelo Inicial). Por exemplo, o relatório das entidades e ações não compartilhadas mostra quais entidades devem ter, necessáriamente, um processo de nível 0 correspondente. Os vários tipos de relatórios de análise deste passo são:

1) Possíveis Entidades Marsupiais: uma entidade pode ser marsupial se ela não tem ações não compartilhadas com outras entidades;

2) Entidades e Ações Não Compartilhadas: para cada entidade, mostra a lista de ações que nenhuma outra entidade sofre ou executa.

A opção "analysis" que aparece no "menu do passo 1" produz um relatório de análise contendo os dois tipos de relatórios citados acima.

4.2.3.2 - PASSO DA ESTRUTURA DA ENTIDADE

Os relatórios de análise produzidos neste passo podem ser obtidos separadamente em duas opções, a serem escolhidas pelo usuário no "menu do passo 2". São elas:

1) Expressões Regulares da Estrutura: para cada estrutura as sequências de ações permitidas formam uma linguagem regular. Este relatório mostra a expressão regular estraída da sequência de ações da entidade;

2) Geração da Sequência da Ação: para cada entidade escolhida, ele gera (aleatóriamente) tantas sequências quantas forem desejadas, de acordo com a definição da estrutura da entidade.

4.2.3.3 - PASSO DO MODELO INICIAL

Os relatórios de análise que podem ser produzidos pelo JSD-tool, para este passo são:

1) Análise da Conectividade: para cada processo P, ele dá todos os outros processos internos que podem ser alcançados por P, e também todos os processos que alcançam o processo P;

2) Expressões Regulares das Sequências de Dados: as expressões regulares que denotam as sequências de dados internas são mostradas neste relatório.

Estes relatórios podem ser obtidos separadamente em duas opções, a serem escolhidas pelo usuário no "menu do passo 3".

4.2.3.4 - PASSO DE FUNÇÃO

Os relatórios de análise que podem ser produzidos pelo JSD-tool, para o passo de função são:

1) Expressões Regulares das Sequências de Dados: como no passo do Modelo Inicial. Aqui, esta análise pode ser especialmente útil para as sequências de dados exclusivamente de entrada;

2) Análise da Conectividade: Como no passo do Modelo Inicial;

3) Dependência das Sequências de Dados de Entrada: para cada sequência da dados de entrada, ele mostra os processos que dependem desta entrada. Esta análise é feita percorrendo o caminho do processo até a entrada.

Estes relatórios podem ser obtidos separadamente em três opções, a serem escolhidas pelo usuário no "menu do passo 4".

4.2.3.5 - PASSO DE SINCRONISMO DO SISTEMA

No modo de análise, apenas estes dois relatórios podem ser produzidos no passo do sincronismo. São eles:

1) Ciclos no DES: lista todos os ciclos que ocorrem no DES;

2) Caminhos de Entrada e Saída: lista todos os caminhos (sem ciclos) das conexões de entrada (enquiry) até as sequências de dados de saída (report).

Estes relatórios podem ser obtidos separadamente em duas opções, a serem escolhidas pelo usuário no "menu do passo 5", ou em uma terceira que produz em um único relatório, o resultado das duas análises referenciadas.

CAPÍTULO 5

EXECUTOR DE ESPECIFICAÇÕES JSD

O método JSD está dividido em duas grandes fases: especificação e implementação. Concluída a primeira fase tem-se, uma especificação constituída do Diagrama de Especificação do Sistema (DES) e da estrutura textual de cada um dos processos especificados. Sob o ponto de vista computacional, a especificação JSD é composta por um conjunto de processos que se comunicam por troca de mensagens ou por inspeção somente de leitura aos dados internos.

Se imaginarmos que existe uma máquina com tantos processadores quantos forem os processos especificados e, ainda, que existe memória em cada processador, suficiente para armazenar todo o texto e o vetor de estados do processo, a especificação (na forma como está descrita no DES) está pronta para ser executada, pois:

1) cada processador pode executar o texto de seu processo associado, comunicando-se com os outros processos e com o mundo externo através das operações de **read** e **write**;

2) o texto do processo (isto é, a estrutura de controle que especifica tanto a ordem das ações quanto as operações executáveis que simulam as ações) e o vetor de estados dos processos (composto pelo conjunto de variáveis internas representando os atributos da entidade) podem ser armazenados internamente ao processador.

Entretanto, estas seriam condições ideais, o que raramente acontece. Uma especificação JSD, frequentemente, possui um número grande de processos e o número de processadores disponíveis, quase sempre, é menor que o número de processos especificados. Pode não haver espaço de memória suficiente para guardar o texto de processo e seu vetor de estados. Assim, o DES deve sofrer algumas transformações para superar as restrições e poder ser executado.

A etapa do desenvolvimento, onde é necessário a introdução de modificações no DES e mesmo na própria descrição dos processos, corresponde à fase de transformação do ciclo de vida do modelo operacional.

Como descrito na Seção 2.2, estas modificações são feitas manualmente, antes da fase de realização, pois a automação das decisões desta fase pode levar a um sistema pouco otimizado e até mesmo indesejável.

As questões abaixo devem ser resolvidas para que se possa implementar o sistema especificado:

a) quais operações devem ser inseridas na descrição do DES e na descrição dos processos para que eles possam ser devidamente executados por um computador;

b) como compartilhar os processadores disponíveis entre os vários processos especificados, permitindo a concorrência dos processos do DES;

c) como permitir que múltiplos processos compartilhem o mesmo código;

d) que técnica usar para implementar a comunicação dos processos através de conexões por sequência de dados;

e) como armazenar e implementar o mecanismo de comunicação por vetor de estados;

f) como implementar multiplicidade de conexões por sequência de dados e por vetor de estados;

g) para qual linguagem de programação traduzir a especificação JSD para que ela possa ser executada;

h) como executar um programa cujo texto não cabe totalmente na memória principal;

i) onde armazenar informações recebidas por um processo, que devem ser processadas depois de algum tempo e não pertencem ao vetor de estados do mesmo;

j) como viabilizar as entradas e saídas do sistema especificado;

l) como garantir que o sistema não entre em deadlock.

Algumas soluções para estas questões são discutidas e resolvidas pelo Jackson, no passo 6 de seu método JSD, como descrito na Seção 3.2.6.

Com o objetivo de automatizar o passo de implementação do método JSD e, com isso, testar a viabilidade de tornar uma especificação executável, foi construída uma máquina abstrata capaz de executar uma especificação elaborada segundo o método JSD.

As soluções adotadas para contornar as restrições de recursos e resolver as questões citadas acima estão descritas na Seção 5.1.

A Seção 5.2 apresenta uma série de restrições que devem ser satisfeitas por uma especificação para que ela seja executada pelo executor-JSD.

Os cuidados tomados na construção desta ferramenta, bem como, a técnica usada para facilitar a introdução de novos comandos e a própria manutenção da mesma, são relacionados na Seção 5.3.

5.1 - SOLUÇÕES ADOTADAS NO EXECUTOR-JSD

O primeiro passo para a construção do executor-JSD foi definir uma sintaxe para a linguagem de especificação baseada no método JSD, a qual fosse capaz de descrever o DES e a estrutura textual dos processos alterando o menos possível a sintaxe da linguagem usada no JSD-tool, a qual cobre todos os recursos do método JSD. Ela é capaz de expressar os dois tipos de conexões (sequência de dados e vetor de estados), a multiplicidade de processos e conexões, uma fusão forçada de entradas de um processo, as conexões de entrada e saída e a estrutura textual dos processos. A descrição dos processos é feita através dos componentes básicos de programação estruturada (sequência, seleção, iteração) e das operações de leitura e escrita a sequência de dados e consulta a vetor de estados.

Além de descrever o comportamento concorrente do DES e o comportamento sequencial de cada um dos processos, a nova linguagem provê comandos para permitir a alocação dos processos e suas variáveis internas, enriquecer as declarações que expressam o comportamento sequencial de cada processo, aumentando assim, o poder expressivo da estrutura textual sob o ponto de vista da máquina.

5.1.1 - COMANDOS DE ESPECIFICAÇÃO

Seguindo a mesma filosofia definida no modo de especificação do JSD-tool, onde os comandos de especificação de um passo posterior englobam os comandos do passo anterior, o conjunto de comandos definidos para cobrir o passo 6 inclui aqueles definidos nos passos três e quatro do JSD-tool. São eles:

a) **spec** - define todos os processos do sistema. Para cada processo especifica as variáveis e o número de instâncias;

b) **sequence** - define um nome de sequência como uma sequência de itens. Cada item pode ser o nome de uma estrutura (isto é, o nome de um sequência, de uma seleção ou de uma iteração). O nome da estrutura pode ser o nome de uma ação ou não. O item pode ser também uma declaração de entrada e saída, uma declaração de atribuição ou, simplesmente, um nome de estrutura. As declarações de entrada e saída compreendem os comandos: **read** - comando para ler uma conexão por sequência de dados e atribuir os valores lidos a uma lista de variáveis; **write** - comando para escrever os valores das variáveis de uma lista em uma sequência de dados ou para escrever um texto em uma sequência de dados de saída; **getsv** - comando para ler uma conexão por vetor de estados e atribuir os valores obtidos a uma lista de variáveis;

c) **selection** - define um nome de seleção como uma sequência de itens. Expressões booleanas devem ser atribuídas a cada parte da seleção. É executada somente a parte da seleção que primeiro contiver uma expressão com resultado verdadeiro;

d) **iteration** - define um nome de iteração como uma lista de itens. Uma expressão booleana deve ser atribuida ao comando definindo assim, a condição de parada da iteração. Instruções **quit** seguidas de expressão booleana podem fazer parte deste comando;

e) **posit** - define um nome de posit seguido de uma lista de itens condicionais, de declarações **quit** e de uma cláusula **admit**;

f) **external stream** - define um nome de conexão entre o mundo real e um processo. O mundo real, na implementação deste comando, é representado pelo monitor de vídeo;

g) **stream** - define um nome de conexão por sequência de dados entre dois processos, e estabelece a relação de multiplicidade da sequência de dados;

h) **state** - define um nome de conexão por vetor de estados, e estabelece a multiplicidade do vetor de estados;

i) **enquiry** - define um nome de conexão entre o mundo real e um processo. Na implementação deste comando, o mundo real é representado pelo monitor de vídeo;

j) **time** - define uma sequência de dados que gera uma entrada de tempo a um processo interno. No caso, o "time" foi implementado através de uma pergunta sobre o tempo feita ao usuário via o terminal de vídeo. A resposta normalmente é inserida em uma entrada com fusão forçada;

l) **report** - define uma conexão entre um processo e o mundo real. No caso, a saída para o mundo real é feita através de mensagens emitidas no monitor de vídeo, de acordo com a implementação adotada para este comando;

m) **rough merge** - indica as sequências de dados que compõem uma fusão forçada;

n) **end** - define a última instrução da estrutura textual de um processo.

A sintaxe da linguagem para definição de especificações JSD, capaz de ser executada pelo executor-JSD, é apresentada no Apêndice B.

5.1.2 - ESCALONAMENTO DOS PROCESSOS

O Executor-JSD foi criado para executar especificações em microcomputadores com um único processador e sem recursos para multiprogramação, logo ele deve conter um mecanismo capaz de rodar os processos concorrentes em diversas máquinas virtuais. O mecanismo de suspende/continua adotado que permite a execução (virtual) de concorrência é o seguinte:

. um processo que não está em execução pode estar em um dos dois estados: suspenso ou pronto;

. um processo em execução, roda até encontrar uma instrução "**read** A to (a)", onde A é uma sequência de dados que está vazia, aí ele passa para o estado de suspenso;

. quando algum processo em execução escreve uma mensagem na sequência de dados A, o processo suspenso que esperava tal recurso (uma mensagem de A) passa ao estado de pronto.

Há uma fila chamada fila de prontos, onde são inseridos os processos que estão prontos para execução e outra fila, chamada fila de suspensos, onde são mantidos todos os processos que estão a espera de uma condição para que possam continuar sua execução.

Um processo nunca sai do estado de suspenso e vai para a execução. Ao sair da fila de suspensos o processo é inserido no final da fila de prontos. De forma que não há prioridade entre eles, isto é, não existe comando que expresse a prioridade de um processo. Quando os processos são declarados no comando **spec** eles são inseridos na fila de prontos, na mesma ordem em que foram declarados.

O processo escalonado é sempre o primeiro da fila de prontos. A ordem original dos processos na fila de prontos, é quebrada, somente, quando um processo volta a esta fila após ficar suspenso por um tempo indeterminado.

Na fila de suspensos, os processos são inseridos sempre no final, porém, não há ordem de saída: sai o processo que primeiro receber o recurso esperado.

O próprio executor engloba a função de suspender e continuar a execução dos processos especificados e desta forma, ele funciona como um escalonador de processos. Por este motivo, às vezes o executor é chamado escalonador.

5.1.3 - PROCESSOS MÚLTIPLOS

O problema dos processos múltiplos que compartilham o mesmo código é resolvido mantendo-se apenas uma cópia do texto e tantas cópias do vetor de estados quantas forem as instâncias do processo, armazenadas na memória principal. As várias instâncias de um mesmo processo são caracterizadas por índices para acesso ao seu vetor de estados. Tanto as instâncias de processo como um processo sem instâncias são vistas pelo escalonador como um processo, cujas informações são mantidas em um "Descritor de Processo" o qual contém:

a) o nome do processo: as várias instâncias de um mesmo processo recebem todas o mesmo nome e são diferenciadas umas das outras pela ordem em que estão posicionadas na Tabela Descritora (nome da tabela que contém os descritores de todos os processos e suas instâncias). Um processo com cinco instâncias terá cinco descritores com o mesmo nome sendo que o primeiro descritor entre os cinco representa a primeira instância do processo, o segundo descritor representa a segunda instância e assim por diante. A ordem da instância não coincide com o índice da tabela e sim com a posição relativa do descritor da instância em relação ao descritor da primeira instância de um processo;

b) o estado do processo: esta informação corresponde a um ponteiro que liga este descritor aos outros descritores da fila de prontos ou da fila de suspensos ou corresponde a um valor nulo caso o processo esteja em execução;

c) o nome da sequência de dados da qual o processo espera uma mensagem quando no estado suspenso: em outras palavras, significa o motivo pelo qual o processo está suspenso. Como aqui um processo só é suspenso a espera de uma mensagem de uma conexão por sequência de dados, este campo pode conter um nome de sequência de dados ou estar vazio;

d) um ponteiro para o vetor de estados: este ponteiro endereça a área que contém o vetor de estados. Cada instância possui uma cópia do vetor de estados. No momento da alocação, todos os vetores de estados de uma mesma instância são idênticos.

Uma outra solução para a multiplicidade dos processos que compartilham o mesmo código seria repetir, além do vetor de estados, também, o código do processo para cada instância. Como o número de instâncias de processo em uma especificação JSD costuma ser muito grande esta possibilidade foi descartada.

Para distribuir melhor as informações internas dos processos, uma possibilidade, seria manter os vetores de estados armazenados em memória secundária cujo acesso poderia ser auxiliado por operações de um banco de dados já existente (caso este recurso estivesse disponível no sistema). Nesta primeira versão do Executor-JSD todos os vetores de estados estão armazenados na memória principal. Ganha-se, com isso, maior velocidade na busca das informações.

5.1.4 - CONEXÃO POR SEQUÊNCIA DE DADOS

O mecanismo de conexão por sequência de dados consta de um processo fonte que escreve mensagens a um processo destino. As mensagens são colocadas em buffers alocados dinamicamente, e inseridos em uma fila através da execução da operação **write** que pertence ao processo fonte da conexão. Na fila, as mensagens são mantidas em ordem de chegada e o processo destino as obtém através da operação **read** na mesma ordem em que foram inseridas. Assim, a primeira mensagem que chega é a primeira a ser lida, este mecanismo é chamado FIFO ("First In, First Out)[1].

Uma sequência de dados é, então, uma fila (FIFO) composta de buffers, onde cada buffer contém somente uma mensagem. Nesta versão, uma mensagem pode ser um caractere alfanumérico ou uma sequência deles.

Este mecanismo parece muito simples, entretanto, o sistema deve ser capaz de executar comunicações que envolvam múltiplos processos, tais como:

---

[1] Primeiro que entra, primeiro que sai.

1) Conexão por sequência de dados do tipo muitos-para-um mostrada na Figura 5.1 de acordo com a notação usada no DES. Este esquema indica que muitas instâncias de um processo P enviam mensagens a um outro processo, chamado Q, que por sua vez não tem instâncias, é único.

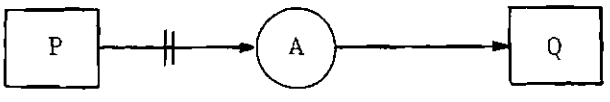


Fig. 5.1 - Conexão por sequência de dados do tipo muitos-para-um.

2) Conexão por sequência de dados do tipo um-para-muitos ilustrada na Figura 5.2 de acordo com a notação usada no DES. Este esquema mostra um processo P, sem instâncias, enviando mensagens a um processo Q, com muitas instâncias.

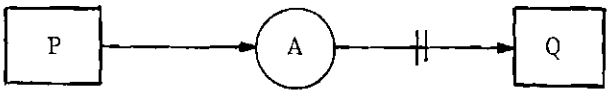


Fig. 5.2 - Conexão por sequência de dados do tipo um-para-muitos.

3) Conexão por sequência de dados do tipo muitos-para-muitos ilustrada na Figura 5.3 de acordo com a notação usada no DES. Neste caso, muitas instâncias do processo P enviam mensagens a muitas instâncias do processo Q.

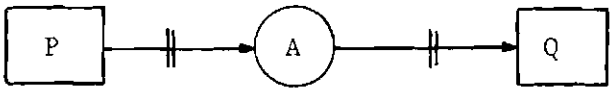


Fig. 5.3 - Conexão por sequência de dados do tipo muitos-para-muitos.

Uma conexão por sequência de dados que envolve múltiplos processos, exige a alocação de múltiplas filas para que seja efetuada a comunicação entre as várias instâncias dos processos conectados. Pois, na conexão por sequência de dados, apenas um processo pode escrever na fila da conexão e apenas um processo pode ler desta[2].

Para calcular o número de filas que devem ser alocadas para implementação da conexão por sequência de dados com multiplicidade de processos, basta multiplicar o número de instâncias do processo fonte com o número de instâncias do processo destino da conexão.

A identificação de uma sequência de dados envolvendo processos múltiplos é feita, então, através de dois índices, um referente a instância do processo fonte e outro referente a instância do processo destino, mantendo-se o mesmo nome da conexão.

Para implementar o acesso à sequência de dados indexada foi necessário o uso de uma matriz **n x m** onde o número de linhas (n) equivale ao número de instâncias do processo fonte e o número de colunas (m) corresponde ao número de instâncias do processo destino da conexão. Cada campo da matriz contém um ponteiro para uma sequência de dados (ou seja, um ponteiro para o topo de uma fila).

Toda sequência de dados é acessada indiretamente através desta matriz. É ela quem mantém a informação da multiplicidade da conexão. Uma conexão que não envolve multiplicidade é representada por uma matriz **1 x 1**.

O tipo de conexão mostrado na Figura 5.1, sob o ponto de vista da implementação, é visto segundo o esquema apresentado na Figura 5.4.

---

[2] O termo **processo**, significa tanto uma instância de processo como um processo único.

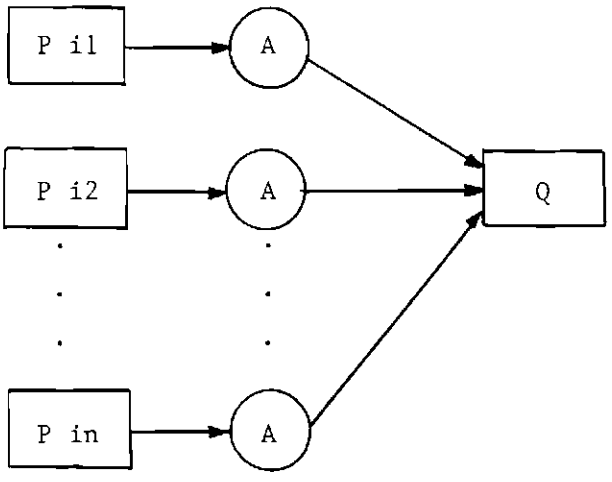


Fig. 5.4 - Conexão por sequência de dados do tipo muitos-para-um para implementação.

Neste tipo de conexão cada instância do processo P está ligada ao processo Q por uma fila. O número de filas é igual ao número de instâncias do processo P, o processo fonte da conexão. Cada instância de P deve conter uma instrução "**write** (a) to A". Na execução desta instrução, em cada uma das instâncias, o valor da variável **a** é escrito em uma sequência de dados diferente.

O processo Q, por sua vez, deve ser capaz de ler as mensagens de todas as filas envolvidas na conexão. Ele deve conter instruções do tipo "**read** A(1) to (q)", "**read** A(2) to (q)",..., "**read** A(n) to (q)", em sua estrutura textual para que se efetue a comunicação com todas as instâncias de P, como indicado na Figura 5.1.

O identificador da sequência de dados A da instrução "**write** (a) to A" não desapareceu. É desnecessário explicitá-lo, pois, durante a execução de um comando de escrita, o identificador não declarado recebe o número de ordem da instância (primeira, segunda, terceira, etc) que está sendo executada. Neste exemplo, quando a k-ésima instância de P (Pik), para (1 < = k < = n), estiver em execução a instrução de escrita será igual a "**write** (a) to A(k)".

Na instrução de leitura do processo Q, o identificador deve ser explícito, pois este corresponde à instância de P da qual o processo Q deve receber mensagens. Se o identificador não estiver explícito, Q receberá mensagens somente da instância Pil, onde o número 1 significa a instância do próprio Q e não de P, logo haverá erro na execução.

O esquema mostrado na Figura 5.2, sob o ponto de vista da implementação fica como ilustrado na Figura 5.5.

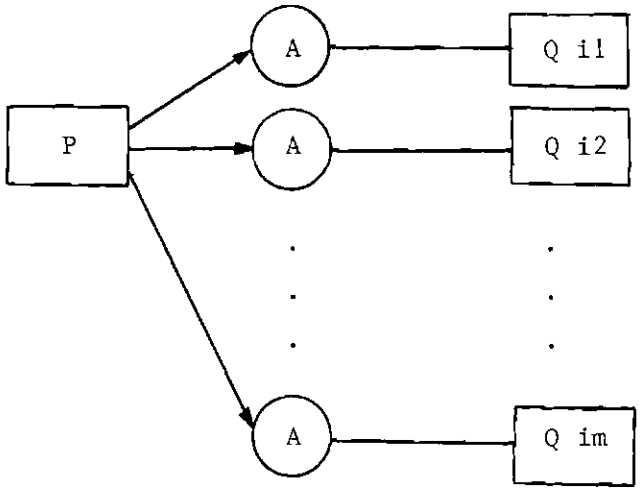


Fig. 5.5 - Conexão por sequência de dados do tipo um-para-muitos para implementação.

Neste tipo de conexão o processo P está ligado a cada instância do processo Q através de uma fila distinta. O número de filas envolvidas na conexão é igual ao número de instâncias do processo Q. O processo P deve conter, em sua estrutura textual, instruções do tipo "**write** (a) to A(identificador)", endereçadas a todas as instâncias do processo Q. Desta forma "identificador" corresponde a números entre **1** e **m** e o valor da variável **a** é escrito em todas as filas sem distinção.

Por outro lado, cada instância do processo Q deve conter pelo menos uma operação "**read** A to (q)" em sua estrutura textual. Aqui, o identificador de A é desnecessário, pois, durante a execução de uma operação de leitura de uma instância, o identificador não explícito recebe um valor igual ao número de ordem da instância. Assim, na k-ésima instância, para (1 < = k < = m), esta instrução é executada como "**read** A(k) to (q)". Como todas as instâncias de um processo devem ser executadas, todas as sequências de dados da conexão serão lidas pelo processo destino.

A conexão apresentada na Figura 5.3, sob o ponto de vista da implementação pode ser esquematizada como ilustrado na Figura 5.6.

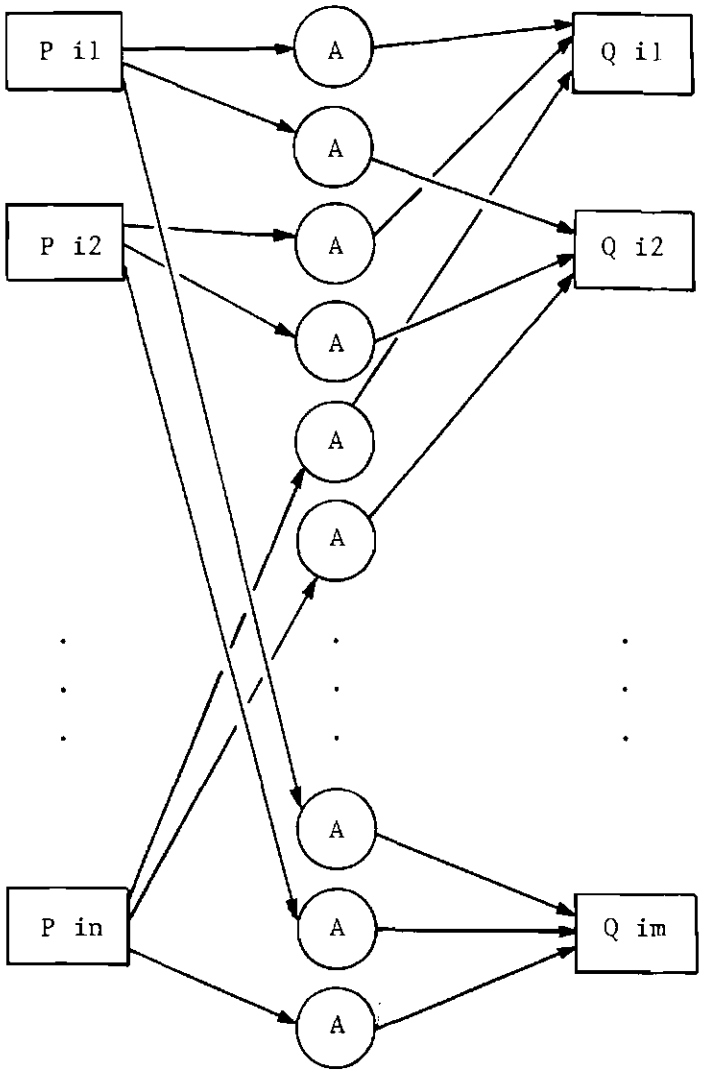


Fig. 5.6 - Conexão por sequência de dados do tipo muitos-para-muitos para implementação.

Na execução deste tipo de conexão (n x m) filas devem ser alocadas para efetuar a comunicação corretamente. Cada instância do processo P se comunica com cada instância do processo Q através de uma fila diferente. As filas podem ser identificadas como A(i,j) onde i é o número da instância do processo fonte e j é o número da instância do processo destino da conexão.

Cada instância do processo P deve conter uma instrução "**write** (a) to A(identificador)" endereçadas a todas as instâncias do processo Q. O valor do identificador corresponde à identificação da instância do processo destino, logo deve variar de 1 a **m**. A identificação da instância do processo fonte está implícita na execução do comando: ela corresponde ao número de ordem da instância que está em execução. Quando a operação "**write** (a) to A(j)" para (1 <= j <= m) da instância k do processo P, onde k é um valor entre 1 **e n**, for executada, as sequências de dados A(k,1), A(k,2), ..., A(k,m) receberão o valor da variável **a**. Se todas as instâncias de P são executadas, k percorre o intervalo (1,..,n) e assim todas as sequências de dados são atualizadas com o valor de **a**.

Do lado do processo Q, cada instância de Q deve conter um operação "**read** A(identificador) to (q)". Aqui, o identificador caracteriza as instâncias de P. O valor do identificador varia dentro do intervalo (1,..,n). Quando a operação "**read** A(i) to (q)" para 1 <= i <= n, da instância k do processo Q for executada, as sequências de dados A(1,k), A(2,k), ..., A(n,k) serão lidas pela k-ésima instância de Q. Se todas as instâncias de Q são executadas, k varia no intervalo (1,..,m) e assim todas as sequências de dados da conexão são lidos por Q.

5.1.5 - CONEXÃO POR VETOR DE ESTADOS

Os vetores de estados são vetores alocados dinamicamente, cujo tamanho depende do número de variáveis do processo.

Um vetor de estados é implementado através de um vetor dividido primariamente em dois campos. O primeiro campo contém o ponteiro para o topo da pilha do processo, a qual contém o endereço da próxima instrução a ser executada, ou seja, o ponteiro de texto do processo. O segundo campo contém o conjunto de variáveis internas. Este campo, é subdividido em tantos segmentos quantas forem as variáveis do processo. Cada segmento contém o nome e o último valor atribuído à variável.

O mecanismo de manter uma pilha com as instruções a serem executadas, permite ao executor continuar a execução de um processo que fora suspenso, a partir do ponto onde ele parara.

O acesso as informações de um vetor de estados pelo processo destino de uma conexão é feito através da execução da operação **getsv**. Após esta operação, o processo destino fica com uma cópia de tantas variáveis do processo fonte quantas ele desejar. Este desejo é expresso através da lista de variáveis que faz parte do comando. Se o número de variáveis do comando for menor que o número de variáveis do vetor de estados então, o processo destino obterá somente as primeiras variáveis do processo fonte. O número de variáveis obtidas equivale ao número de espaços alocados para armazenamento no processo destino.

Desta forma, o processo destino da conexão fica com uma cópia das informações que lhe interessam em sua própria área de dados. Estas informações correspondem ao valor das variáveis do processo fonte no momento da execução da operação **getsv**. É garantido que o processo fonte não estava alterando o valor destas informações, dado que há um único processo em execução sobre o único processador disponível, no caso da execução através do Executor-JSD. Não podendo ser garantido para o caso geral de multiprogramação.

Nas conexões por vetor de estados não há dois tipos de operações envolvidas na comunicação, apenas uma, a operação **getsv** que pertence à estrutura textual do processo destino da conexão.

Uma conexão por vetor de estados também pode envolver múltiplos processos, ou seja, ela pode ser do tipo muitos-para-um, um-para-muitos, muitos-para-muitos:

1) A conexão por vetor de estados do tipo muitos-para-um é mostrada na Figura 5.7, de acordo com a notação usada no DES. Este esquema mostra um processo Q, sem instâncias, obtendo informações das várias instâncias do processo fonte da conexão, chamado P.

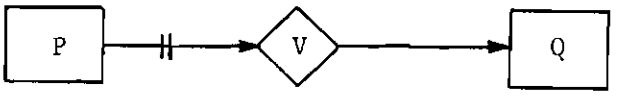


Fig. 5.7 - Conexão por vetor de estados do tipo muitos-para-um.

2) A conexão por vetor de estados do tipo um-para-muitos é ilustrada na Figura 5.8, de acordo com a notação usada no DES. Este esquema mostra que várias instâncias do processo destino da conexão, chamado Q, podem obter informações do vetor de estados do processo P (sem instâncias).

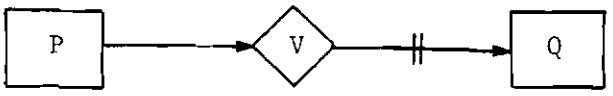


Fig. 5.8 - Conexão por vetor de estados do tipo um-para-muitos.

3) A conexão por vetor de estados do tipo muitos-para-muitos é ilustrada na Figura 5.9, de acordo com a notação usada no DES. Este esquema ilustra uma situação onde as várias instâncias do processo destino da conexão, chamado Q, podem obter informações do vetor de estados de cada instância do processo P.

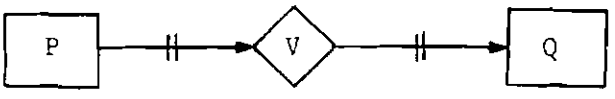


Fig. 5.9 - Conexão por vetor de estados do tipo muitos-para-muitos.

Uma conexão por vetor de estados que envolve múltiplos processos não exige alocação especial de vetores de estados para a comunicação. O número de vetores alocados corresponde, sempre, ao número de instâncias do processo fonte da conexão e estes são alocados na execução da primeira instrução de um especificação. Como os vetores de estados são formados basicamente pelas variáveis internas dos processos, eles são criados na execução do comando **spec**, isto é, quando da declaração das variáveis do processo. Neste mesmo comando é especificado o número de instâncias do processo e assim são criados tantos vetores de estados quantas forem as instâncias.

O tipo de conexão mostrada na Figura 5.7, sob o ponto de vista da implementação, é visto como o esquema apresentado na Figura 5.10.

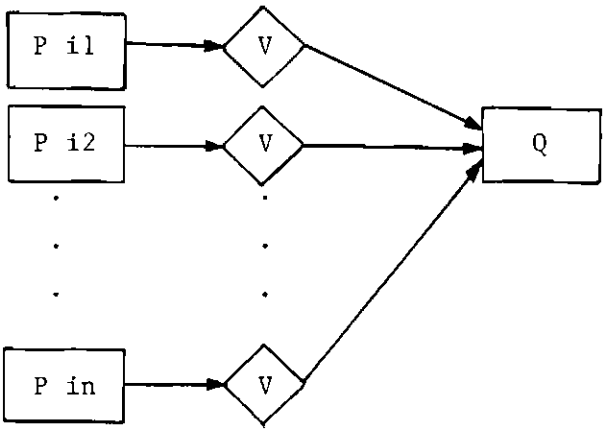


Fig. 5.10 - Conexão por vetor de estados do tipo muitos-para-um para implementação.

Neste tipo de conexão, o processo Q deve conter em sua estrutura textual a operação "**getsv** V(i) to (q)". O identificador i que acompanha o nome da conexão, V, é necessário para indicar exatamente o vetor de estados a ser obtido. No caso, o vetor de estados pertence a i-ésima instância de P. O processo P é considerado passivo nesta conexão, pois ele não executa alguma operação e não sofre consequências com a execução de **getsv**, para o caso da execução através do Executor-JSD. Não podendo ser garantido no caso geral de multiprogramação.

A conexão do tipo um-para-muitos mostrada na Figura 5.8, sob o ponto de vista da implementação, fica como ilustrado na Figura 5.11.

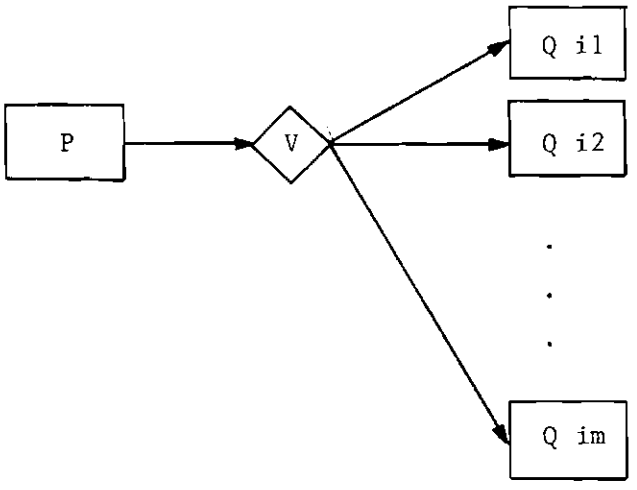


Fig. 5.11 - Conexão por vetor de estados do tipo um-para-muitos para implementação.

Neste tipo de conexão as instâncias do processo Q devem conter em sua estrutura textual, uma operação "**getsv** S to (q)". Aqui não é necessário explicitar a identificação da conexão dado que há apenas um vetor de estados a ser consultado, o vetor de estados de processo P, único.

A conexão do tipo muitos-para-muitos mostrada na Figura 5.9, sob o ponto de vista da implementação, é vista como o esquema apresentado na Figura 5.12.

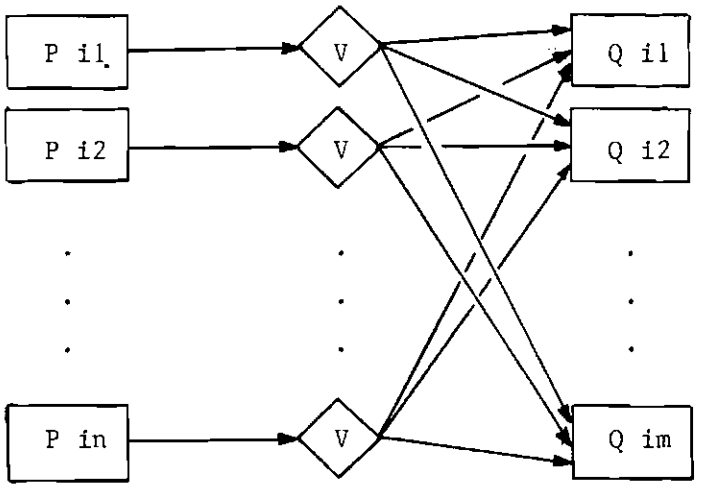


Fig. 5.12 - Conexão por vetor de estados do tipo muitos-para-muitos para implementação.

Neste tipo de conexão, o número de vetores de estados envolvidos é igual ao número de instâncias do processo P. Todas as instâncias de Q podem obter os vetores de todas as instâncias de P. Para isso, cada instância de Q deve conter uma operação "**getsv** V(i) to (q)" em sua estrutura textual. Quando esta operação é executada o valor de i pode variar de 1 a n. (onde **n** é o número de instâncias de P). Se isso acontece cada instância de Q obtém os valores do vetor de estados de todas as instâncias de P. O identificador **i** é exigido no comando "**getsv** V(i) to (q)" para permitir que uma dada instância de Q, por exemplo, a k-ésima, possa acessar o vetor de estados da **i**-ésima instância do processo P. Cabe observar que o identificador i corresponde ao índice da instância do processo fonte da conexão.

5.1.6 - ENTRADAS E SAÍDAS

A comunicação entre o sistema especificado e o mundo externo, feita através de conexões por sequência de dados, foi implementada no Executor-JSD, como descrito nos parágrafos abaixo.

As entradas do sistema podem ser feitas através de instruções de leitura a um dos tipos de sequência de dados: externa, enquiry, e marcador de tempo, definidas pelos comandos **external stream**, **enquiry** e **time**, respectivamente. As sequências de dados definidas por estes comandos, chamadas sequências de dados de entrada apresentam diferenças entre elas, mas, aqui todas comportam-se da mesma maneira.

O Executor-JSD ao encontrar uma instrução de leitura a uma sequência de dados de entrada mostra uma mensagem no vídeo, a qual solicita ao usuário um valor para uma dada sequência de dados. Enquanto o usuário não teclar um valor seguido de uma tecla "enter" o processo fica no estado de execução na tentativa de concluir a execução do comando de leitura. Sob o ponto de vista do processamento, este processo está bloqueando a CPU, impedindo que outros processos sejam executados enquanto a mensagem esperada não chega.

A fim de evitar uma espera ocupada da CPU e simular a possibilidade de uma sequência de dados de entrada estar vazia, o executor é munido de um filtro para mensagens externas. Ao reconhecer a mensagem "-", o escalonador suspende o processo em execução, que esperava uma mensagem externa e escalona um outro processo.

No caso da conexão via marcador de tempo, o usuário pode digitar "-" enquanto não for atingido o ponto esperado no tempo, por exemplo, o fim de um minuto, ou o fim de um dia, etc. Uma outra possibilidade é digitar um valor nulo. Esta última é mais indicada quando o marcador de tempo faz parte de um fusão forçada, assim evita-se a suspensão do processo. Estas alternativas dependem da escolha de analista e do problema a ser resolvido.

As saídas do sistema são implementadas através da execução de instruções de escrita a sequência de dados do tipo "report" ou "sequência de dados externa". Ao encontrar uma instrução desta, o executor envia ao terminal de vídeo o conteúdo das variáveis ou a mensagem que fizer parte do comando "write".

Um comando de escrita a uma sequência de dados que não for de saída, provoca a escrita de uma mensagem a uma fila alocada na memória principal. Porém, se a sequência de dados for de entrada é emitida uma mensagem de erro e a execução é encerrada.

A comunicação por vetor de estados entre o sistema especificado e o mundo real não foi implementada nesta versão do Executor-JSD.

5.1.7 - FUSÃO FORÇADA E OUTROS DETALHES

A fusão forçada é um recurso do método JSD que permite a um processo receber mensagens de várias sequências de dados como se elas chegassem por uma única entrada, onde é mantida a ordem de chegada das mensagens.

A fusão forçada é declarada na especificação como uma lista de sequência de dados.

As sequências de dados que fazem parte de uma fusão forçada correspondem todas a uma única fila, sob o ponto de vista do processo destino da conexão. Porém, cada um dos processos fontes vê a sequência de dados como uma fila de sua propriedade. Sob o ponto de vista da implementação, a comunicação consta de uma única fila, na qual muitos processos podem escrever mas apenas um pode ler.

Como foi citado na Seção 5.1.4, a implementação do acesso a uma sequência de dados é composta de uma matriz associada a ela[3], na qual cada campo aponta para uma fila diferente. Nas conexões com fusão forçada todas as matrizes, cada qual associada a uma sequência de dados pertencente a fusão forçada, apontam para uma única fila (FIFO).

---

[3] Matriz **nxm** em conexões envolvendo multiplicidade de processos ou Matriz **1x1** em conexões envolvendo um único processo.

Assim, é mantida a ordem de chegada das mensagens, os processos fontes imaginam escrever cada um em uma fila (pois a escrita é feita a sequências de dados com nomes diferentes) e o processo destino obtém as mensagens de todos os processos envolvidos na conexão.

Outras particularidades definidas pelo método JSD no passo 6 tais como o desmembramento de processos, a criação de buffers para permitir que um processo armazene informações temporariamente não são implementadas nesta versão.

O cuidado para evitar que o sistema especificado não entre em deadlock deve ser uma preocupação do analista na fase de transformação da especificação, portanto antes da fase de realização.

Na fase de realização de um método não automatizado, a especificação do sistema deve ser traduzida para uma linguagem de programação, entretanto, este não é o caso das especificações escritas na linguagem de specificação JSD definida no Apêndice B. Esta linguagem é formal, existe um interpretador (dentro do Executor-JSD) capaz de entendê-la, assim a própria especificação pode ser executada.

5.2 - LIMITAÇÕES DO EXECUTOR-JSD

As restrições do Executor-JSD são relacionadas nesta seção. Porém, não são levantadas as restrições intrínsecas ao método JSD, somente as limitações do executor com relação ao método. Dado que a ferramenta é demonstrativa, as soluções adotadas foram as mais simples e nem sempre as mais eficientes. Algumas soluções já previstas são também discutidas nas seções posteriores.

5.2.1 - COMANDOS DE ESPECIFICAÇÃO

Os comandos de especificação definidos cobrem todas as necessidades de um especificação JSD. Entretanto, nesta versão de Executor-JSD, alguns comandos apresentam restrições no momento da execução, outras limitações são referentes aos comandos definidos ou necessários a especificação para que ela possa ser executada. Tais limitações são apresentadas nas próximas seções.

Uma atribuição pode conter operações aritméticas. Porém, não é permitido o uso de operadores unários.

Não há comandos que permitam a declaração de tipos de variáveis, como, inteiro, real, booleano, caractere, sequência de caracteres, variáveis indexadas, isto é, vetores, matrizes, e outras.

Todas as variáveis do sistema devem pertencer a um processo. Não há comandos para a declaração de variáveis globais, logo, elas não podem existir na especificação.

Os comandos que envolvem a técnica de backtracking definidos no método JSD (posit, quit, e admit) são aceitos pela sintaxe da linguagem porém, a execução destes comandos ainda não foi implementada.

Os vários tipos de conexões de entrada permitidos pelo método JSD também são permitidos pela sintaxe da linguagem de especificação definida aqui. Contudo, a execução de uma instrução de leitura a qualquer um dos tipos de sequência de dados de entrada é executada da mesma forma. Na execução de um comando de leitura não existe diferença entre uma sequência de dados definida em um comando "external stream" e outra definida em um comando "enquiry". Da mesma forma, não existe diferença entre uma sequência de dados definida em um comando "external stream" e um comando "report", na execução de um comado de escrita a uma sequência de dados destes tipos.

5.2.2 - ESCALONAMENTO DOS PROCESSOS

O mecanismo de suspensão/continuação que é responsável pelo escalonamento dos processos é baseado somente na operação de leitura, isto porque as filas que implementam as sequências de dados são de tamanho ilimitado, sendo preenchidos enquanto há espaço de memória disponível. Cabe ao analista que especifica o sistema evitar processos produtores que geram mensagens indefinidamente, pois quando há insuficiência de memória a execução é encerrada.

Uma possibilidade para melhorar esta situação é implementar a sequência de dados como uma fila circular de tamanho limitado. Este fato provocará uma alteração no escalonamento dos processos pois, um comando de escrita poderá, também, suspender um processo, caso a fila esteja cheia no momento da execução do comando write.

A sincronização entre os processos através dos recursos de comunicação disponíveis no método (sequência de dados e vetor de estados) deve ser uma preocupação da fase de especificação, isto é, do analista e não da fase de implementação. O executor não oferece auxílio ao usuário neste aspecto.

Não é permitido especificar tempo para execução dos processos. Há outras linguagens de especificações executáveis, como por exemplo o PAISLey, onde é permitido ao usuário estabelecer um tempo mínimo para a execução de determinados caminhos na especificação.

5.2.3 - PROCESSOS MÚLTIPLOS

Todas as tabelas utilizadas pelo Executor-JSD são de tamanho limitado. O limite no tamanho da Tabela Descritora causa uma restrição no número de processos especificados.

Um processo múltiplo é caracterizado pelas suas instâncias. O número de instâncias de um processo, a princípio, não é limitado. Porém, cada instância possui um descritor de processo e estes são alocados na Tabela Descritora juntamente com os descritores dos processos únicos. Desde que a soma do número de instâncias de todos os processos especificados, não ultrapasse o número de descritores permitidos na Tabela Descritora, o número de instâncias de um processo não é limitado.

O número de instâncias de um processo não pode ser modificado durante a execução de uma especificação. Alterar o número de instâncias implica em alterar o comando **spec**. Quando a especificação sofre uma modificação qualquer, esta especificação deve ser executada novamente, para se obter os novos resultados.

5.2.4 - CONEXÃO POR SEQUÊNCIA DE DADOS

Uma sequência de dados é implementada por uma lista linear simplesmente encadeada, onde seus elementos são alocados dinamicamente durante a execução de um comando de escrita. Não há controle do tamanho desta fila, ela pode crescer infinitamente. Caso haja um processo no sistema especificado, que possa escrever ininterruptamente em uma fila, este processo poderá esgotar os buffers do sistema.

A alocação dos buffers que compõem as filas das sequências de dados é feita pelo alocador de memória que consta das rotinas **malloc(n)** e **free(p)** da própria linguagem C (usada na programação do executor). A rotina malloc(n) retorna um ponteiro **p** para **n** posições consecutivas e free(p) devolve a área de memória adquirida assim ela pode ser reusada mais tarde (KERNIGHAN, 1978).

Em alguns compiladores estas rotinas são bastante rudimentares exigindo que o último buffer fornecido seja o primeiro a ser devolvido para que este possa ser reutilizado. O compilador TURBOC usado no desenvolvimento do executor apresentou problemas na execução da rotina free. Como a especificação é concorrente não há como controlar a ordem de solicitações/devoluções dos buffers ao compilador. Uma solução seria manter o controle de alocação de memória para as sequências de dados, sob controle do próprio executor.

Entretanto, esta é uma restrição que pode ser facilmente contornada, bastando obter uma versão mais poderosa do compilador e fazer o processo destino devolver o buffer da mensagem logo após consumí-la (rotina esta que já está implementada). Por hora, uma especificação ao ser executada corre o risco de ser interrompida por falta de memória, não por estar mal dimensionada, mas por possuir uma quantidade de troca de mensagens maior do que o sistema pode suportar.

5.2.5 - CONEXÃO POR VETOR DE ESTADOS

Um vetor de estados armazena todas as variáveis do processo ao qual ele pertence. Desta forma, o número de variáveis de um processo é limitado devido ao limite no tamanho do vetor de estados.

As variáveis de um processo podem ser de apenas três tipos: um inteiro, um caractere e uma sequência de caracteres (string). No vetor de estados todas as variáveis são caracterizadas por dois ponteiros. Um deles aponta para uma área que contém o nome da variável e o outro aponta para uma área que contém o seu valor. Não há mais informações armazenadas no vetor de estados sobre as variáveis. O tipo da variável é descoberto pelo executor ao acessar a área onde ela está armazenada.

O usuário deve estar atento ao manipular as variáveis para não usar uma variável contendo um ou mais caracteres em uma operação aritmética. As expressões booleanas são permitidas através da comparação do conteúdo de duas variáveis.

5.2.6 - ENTRADAS E SAÍDAS

Os limites mais facilmente visualizados do Executor-JSD são os de entrada e saída do sistema, dado que, todas as entradas e todas as saídas de todas as instâncias de processo são efetuadas sobre o único terminal de vídeo do microcomputador.

A única vantagem desta limitação é que o usuário pode ver toda a concorrência e acompanhar o desempenho do sistema simplesmente olhando para o vídeo.

As entradas e saídas do sistema, ou seja, a comunicação do sistema com o mundo externo, no método JSD, podem ser feitas através de conexões por sequência de dados ou por vetor de estados. Porém, o Executor-JSD não possui recursos para executar conexões externas por vetor de estados. Um processo não pode obter o vetor de estados de uma instância do mundo real e também não é permitida a uma entidade externa obter o vetor de estados de um processo de máquina.

A sequência de dados de entrada de tipo "time" espera sempre um comando externo. Uma outra forma de implementá-la seria, ao encontrar um comando "**read** marcador de tempo", executar uma leitura ao relógio do sistema. Esta solução, porém, seria limitante também, pois não permitiria a suspensão do processo, haveria um limite no tipo de informação obtida, sempre um valor igual a hora, minuto e segundo e não simplesmente uma marca ou um sinal.

Um comando de escrita em uma sequência de dados do tipo "report" poderia ser uma saída na impressora do micro, dado que, esta saída é sempre uma saída do tipo relatório, ela estaria sempre registrada no papel.

5.2.7 - FUSÃO FORÇADA E OUTROS DETALHES

O problema de uma fusão forçada surge quando há uma sequência de dados do tipo marcador de tempo que faz parte da fusão.

Na execução do comando "**read** A&B to (a)" somente o endereço da primeira sequência de dados, no caso A, é consultado. A consulta ao endereço de todas as sequências de dados é desnecessária dado que, todas as outras apontam para a mesma fila[4].

---

[4] Dada a forma como a fusão forçada foi implementada.

Porém, se houver um marcador de tempo na fusão forçada [5] surge um problema, pois, um marcador de tempo é uma sequência de dados de entrada cujo valor deve ser buscado no terminal de vídeo (como está descrito na seção anterior), logo, não há uma fila na memória para armazenar as informações desta conexão. Como, então, fazer uma mensagem externa cair em uma fila interna?

Este problema foi contornado da sequinte forma: ao encontrar um comando "**read** A&B to (a)" o executor, verifica se a primeira sequência de dados corresponde a um marcador de tempo. Se for um marcador de tempo, o comando de leitura a uma sequência de dados de entrada é executado, solicitando-se ao usuário o valor do marcador de tempo. Se o valor obtido for igual a "-" , o processo é suspenso a espera de mensagem nesta conexão. Caso contrário, o valor obtido é inserido no final da fila da fusão forçada e o primeiro valor desta fila é atribuído a variável do comando.

A restrição que surgiu aí, é que uma fusão forçada envolvendo um marcador de tempo ou uma outra sequência de dados de entrada, deve ser denominada com o nome da conexão de entrada na primeira parte do nome da fusão forçada, uma vez que o executor não procura uma sequência de dados de entrada nas demais partes do nome da fusão forçada.

O executor não fornece qualquer condição em sua estrutura ou na estrutura interna de cada processo para o armazenamento temporário de informações, ou seja, não é permitido o uso de buffers sugerido pelo Jackson para solucionar alguns problemas particulares na fase de implementação.

---

[5] O que é permitido pelo método JSD.

5.3 - CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO DO EXECUTOR-JSD

O Executor-JSD foi implementado em linguagem C sob o sistema operacional DOS podendo ser executado em qualquer microcomputador IBM-PC compatível. O compilador utilizado foi o TURBOC da Borland (BORLAND, 1978). Esta escolha deveu-se à facilidade apresentada por tal compilador em operar em microcomputador com apenas dois controladores de disco flexíveis comuns, sem a opção de um disco rígido. Dado o limite de recursos disponíveis na época da implementação optamos por este compilador.

A linguagem C foi adotada por ser uma linguagem de programação de propósitos gerais com características de expressão, controle de fluxo e estruturas de dados econômicas. A ausência de restrições e sua generalidade tornam-a mais conveniente e eficaz para muitas tarefas do que outras linguagens supostamente mais poderosas (KERNIGHAN, 1978).

A técnica de programação seguida foi a de abstração de dados, pois, facilita a implementação e a manutenção, melhora a produtividade e a segurança do software, além de permitir a construção de componentes reusáveis.

Uma abstração de dados compreende um grupo de operações ou funções realizadas, que agem sobre uma classe particular de objetos, com a restrição de que o comportamento dos objetos pode ser observado somente pelas aplicações das operações.

O uso de abstrações favorece a boa estruturação do programa. O programa é quebrado em unidades menores e independentes. Em cada unidade pode ser mostrada sua correção separadamente. Assim, a verificação de um programa grande inteiro consiste da verificação de um número de programas pequenos. Desde que cada pedaço seja pequeno, a prova do todo é relativamente simples.

O método adotado na construção do Executor-JSD, usando abstrações, consistiu em identificar um número de abstrações auxiliares que são úteis no domínio do problema; estas abstrações auxiliares constituem uma máquina abstrata fornecendo objetos e operações feitas sob medida para implementar a abstração que está sendo estudada. Uma vez identificadas as abstrações, a documentação se dá por duas partes de informação:

1) o comportamento pretendido de cada abstração. A especificação (este termo significa a definição do comportamento da abstração e não tem o sentido usado nos primeiros capítulos) deve descrever todas as entradas e saídas (implícitas e explícitas) da abstração e, então, descrever completamente como a abstração mapeia entradas em saídas;

2) a decisão de quais abstrações auxiliares serão usadas na implementação de uma abstração de nível mais alto. A relação de dependência entre as abstrações pode ser relatada através de uma estrutura gráfica. Assim, quando uma decisão é mudada, a informação está disponível no grafo do projeto onde é fácil identificar o conjunto de abstrações que são afetadas pela mudança.

Resumidamente, a metodologia consiste em executar quatro passos sobre a abstração:

a) identificar abstrações auxiliares que serão úteis na implementação de uma outra abstração;

b) especificar o comportamento de cada abstração auxiliar;

c) escrever um programa para implementar a abstração corrente em termos das abstrações auxiliares;

d) verificar se o programa implementa corretamente a abstração.

A descrição de uma abstração de dados consiste de um **cabeçalho** que nomeia a abstração de dados e suas operações; um breve comentário sobre a abstração como um todo, e uma descrição de cada operação.

As operações pertencentes a uma abstração de dados são consideradas abstrações de procedimento.

Uma abstração de procedimento aceita um conjunto de entradas e fornece um mapeamento de suas entradas para um conjunto de saídas, possívelmente modificando algumas das entradas. O formato da especificação para uma abstração de procedimento é:

. cabeçalho

. linha de modificações

. linha de requisitos

. linha de efeitos

Onde a primeira linha define a interface externa da abstração, incluindo seu nome, a ordem e o tipo de todas as suas entradas, o nome de seus estados de encerramento e o tipo de saída em cada estado. A linha de modificações define quais entradas podem ser modificadas. A linha de requisitos descreve todos os requisitos do ambiente de chamada. E finalmente, a última linha descreve o comportamneto pretendido da abstração de procedimento. Nem todas as linhas são obrigatórias.

Um grafo formado por nós e arcos, onde cada nó é um retângulo que representa uma abstração de dados ou uma abstração de procedimento, apresenta a relação entre todas as abstrações definidas e serve como parte da documentação do projeto juntamente com a descrição das abstrações.

No grafo, os retângulos que representam uma abstração de dados possem um traço duplo em seu topo para diferenciar as abstrações de dados das abstrações de procedimento que são representadas sem o traço duplo.

5.3.1 - DESCRIÇÃO DAS ABSTRAÇÕES DO EXECUTOR-JSD

A descrição da fase de projeto definido para documentar a implementação do executor, é descrita a seguir. Ela segue a metodologia de programação baseada no reconhecimento de abstrações úteis (LISKOV, 1979), como descrito acima.

O grafo completo das abstrações é ilustrado na Figura 5.13.

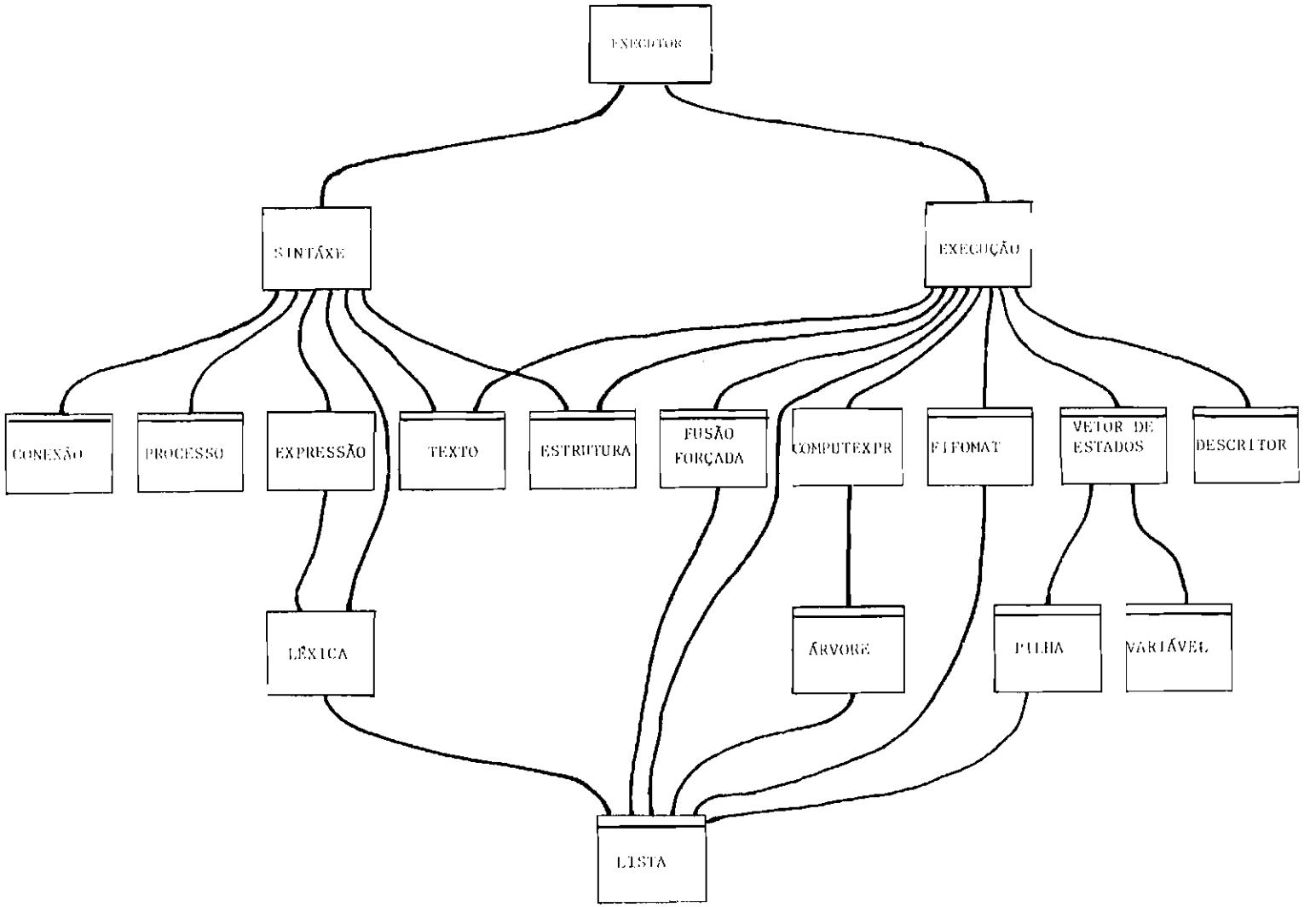


Fig. 5.13 - Grafo de projeto das abstrações do Executor-JSD.

A descrição das abstrações de dados é apresentada em seguida, em ordem alfabética. As abstrações de procedimento são descritas após a descrição das abstrações de dados. O nome das abstrações apresentadas nesta descrição são encontrados traduzidos para o inglês na listagem do programa. Isto porque este programa originou do programa JSD-tool que por sua vez não foi desenvolvido no Brasil. Para manter a mesma apresentação do JSD-tool, todos os nomes de rotinas, variáveis, e abstrações estão em lingua inglesa.

As abstrações de dados do programa Executor-JSD são:

Árvore é create_tree, addl_tree, addr_tree, get_value, tree_left, tree_right, get_string.

Uma árvore é uma estrutura que contém uma expressão aritmética ou booleana. A expressão é formada por operadores e valores inteiros.

create_tree(termtree: inteiro) **retorna**(árvore)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para uma nova árvore. Se não houver memória suficiente, interrompe a execução.

addl_tree(father, son: árvore) **retorna**( )
modifica father
efeito Insere son á esquerda de father.

addr_tree(father, son: árvore) **retorna**( )
modifica father
efeito Insere son à direita de father.

get_value(args: árvore)
retorna(o valor contido no nó da árvore)
efeito Obtém o valor contido no nó da árvore.

tree_left(args: árvore)
retorna(filho esquerdo da árvore)
efeito Obtém o filho esquerdo da árvore.

tree_right(args: árvore)
retorna(filho direito da árvore)
efeito Obtém o filho direito da árvore.

get_string(args: árvore, vstate: vetor de estados)
retorna(o valor da variável contendo uma string)
efeito Obtém o texto da variável cujo nome está em um nó da árvore.

Conexão é checkconnect, addconnect, modconnect, wrconnect.

Conexão é uma tabela onde cada linha representa uma conexão contendo as seguintes informações, nome, tipo, processo fonte, número de processos fontes, processo destino e número de processos destinos da conexão.

checkconnect(entry: inteiro) **retorna**(índice)
sinal(não é conexão)
efeito Retorna o índice da tabela onde encontra-se a conexão. Caso a conexão não seja encontrada retorna zero.

addconnect(name, type, process1, mult1, process2, mult2: inteiro) **retorna**(nconnect)
sinal(tabela cheia)
efeito Insere uma conexão no final da tabela. Caso a tabela esteja cheia retorna um código de ausência de espaço.

modconnect(index, name, type, process1, mult1, process2, mult2: inteiro) **retorna**( )
modifica o conteúdo da linha apontada por index.
efeito Troca os valores de uma conexão na tabela de conexões.

wrconnect(fout: arquivo, level, conn: inteiro) **retorna**( )
efeito Imprime as informações sobre uma conexão.

Descritor é create_descr, ins_ready, ins_susp, rem_ready, rem_susp, create_instances, get_instances, findinst, isprocess, get_sp, get_psv, get_lastinstr, checkdatastr.

Descritor é uma estrutura que contém informações sobre um processo. Um conjunto de descritores são armazenados sequencialmente na Tabela Descritora. Estas informações são: nome, ponteiro para o vetor de estados, ponteiro para ligação em uma das filas de prontos ou de suspensos, o nome de uma sequência de dados caso o processo esteja na fila de suspensos e o endereço da última instrução do processo.

create_descr(index, procname, numvar: inteiro)
retorna(1)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para um novo descritor de processo. Se não houver memória suficiente, interronpe a execução.

ins_ready(procindex: inteiro) **retorna**( )
efeito Insere o descritor de procindex no final da fila de prontos.

ins_susp(dstrname: inteiro) **retorna**( )
efeito Insere o descritor do processo em execução no final da fila de suspensos.

rem_ready( )
retorna(índice do primeiro descritor da fila de prontos)
efeito Remove o primeiro descritor da fila de prontos.

rem_susp(procname: inteiro)
retorna(índice do descritor do processo que foi ativado)
efeito Remove o descritor do processo ativado, da fila de suspensos.

create_instances(procentry, ninst, nvar, varlist:inteiro)
retorna( )
efeito Cria "ninst" descritores de processos iguais, para as instâncias do processo "procentry".

get_instances(procname: inteiro)
retorna(número de instâncias do processo "procname")
efeito Percorre a tabela descritora até encontrar "procname", incrementa um contador até encontrar um valor diferente de "procname".

findinst(procname: inteiro)
retorna(número da instância de processo que está sendo executada)
efeito Contabiliza o número de instâncias desde a que está executando até a última, obtém o número total de instâncias de "procname", subtrai o segundo valor do primeiro.

isprocess(procname: inteiro)
retorna(índice do descritor que contém "procname")
sinals(procname não é um processo)
efeito Percorre a tabela descritora até encontrar um descritor com "procname".

get_sp(procindex: inteiro) **retorna**(pilha do processo)
efeito Obtém o ponteiro do topo da pilha do processo no vetor de estados do mesmo.

get_psv(procindex: inteiro)
retorna(vetor de estados do processo)
efeito Obtém o vetor de estados do processo no descritor do mesmo.

get_lastinstr(procindex: inteiro)
retorna(última instrução do processo)
efeito Obtém a última instrução do processo.

checkdatastr(dstrname: inteiro)
retorna(índice do processo suspenso que espera o recurso "dstrname")
sinal(não há processo à espera de "dstrname")
efeito Percorre a fila de suspensos à procura de "dstrname" no campo "datastr" do descritor.

Estrutura é checkstruct, addstructure, modstructure, wrstructure.

Estrutura é uma tabela que contém as instruções executáveis de todos os processos. Cada linha da tabela contém informações sobre uma instrução. Uma instrução pode ter um nome, um tipo (seq, sel, iter, alt, read, write, getsv, atrib), uma lista de informações úteis à execução.

checkstruct(entry, ityp: inteiro)
retorna(índice da instrução)
sinal(instrução não encontrada)
modifica ityp.
efeito Verifica se "entry" foi definido como uma estrutura.

addstructure(name, ityp: inteiro, items: lista)
retorna(índice da última estrutura inserida)
sinal(não há espaço na tabela)
efeito Adiciona uma estrutura à tabela.

modstructure(index, name, ityp: inteiro, items: lista)
retorna( )
efeito Modifica uma dada estrutura.

wrstructure(fout: FILE, level, struct: inteiro)
retorna( )
efeito Escreve estruturas.

Fifomat é create_fifomat, modfifomat, get-fifo.

Fifomat é uma matriz cujo número de linhas corresponde ao número de instâncias do processo fonte da conexão e o número de colunas coresponde ao número de instâncias do processo destino. Cada elemento da matriz é um ponteiro para uma fila do tipo "FIFO".

create_fifomat(nump1, nump2: inteiro)
retorna(fifomat com dimensão nump1 X nump2)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para um fifomat nump1 X nump2.

modfifomat(fifomat: fifomat, i1, i2: inteiro, fifo:lista)
retorna( )
efeito Insere um ponteiro para uma fila "fifo" em uma dada posição da matiz.

get_fifo(fifomat: fifomat, i1, i2: inteiro)
retorna(fila "fifo")
efeito Obtém o valor de uma posição da matriz de fifos.

Fusão forçada é addrough, wrrough, find_rough, ins_rough.

Fusão forçada é um vetor que contém em cada elemento uma lista de todas as sequências de dados que pertencem a uma fusão forçada.

addrough(rmerge: lista)
retorna(índice da última posição onde foi inserida a fusão forçada)
sinal(não há espaço na tabela)
efeito Adiciona uma lista de sequência de dados à última posição livre do vetor.

wrrough(level, irough: inteiro, fout: FILE) **retorna**( )
efeito Escreve as fusões forçadas já definidas.

find_rough(dstrname: inteiro)
retorna(lista de sequência de dados)
sinal(sequência de dados não pertence a uma fusão)

efeito Verifica se uma dada sequência de dados pertence a alguma fusão forçada definida no sistema. Se pertence retorna o ponteiro para a lista das sequências de dados de tal fusão.

ins_rough(lrough, dstr: lista) **retorna**( )
efeito Insere um valor na sequência de dados de uma fusão forçada.

Lista é listalloc, listfree, mklist, insert, concat, poplast, firstelt, push, include, pop, belong, delelt, equal, setequal, delete, merge, enter, setdif, copy, prlist.

Lista é uma lista simplesmente encadeada, onde cada elemento tem dois campos. Um campo possui um valor inteiro e o outro possui um ponteiro para o próximo elemento da lista.

listalloc( ) **retorna**(lista)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para um nó da lista. Se não houver espaço suficiente, interrompe a execução.

listfree(listnode: lista) **retorna**( )
efeito Libera espaço de um nó da lista.

mklist(element: inteiro) **retorna**(lista)
efeito Aloca espaço para um nó da lista. Insere "elemento" no nó alocado. Se não houver espaço suficiente, interrompe a execução.

insert(list: lista, element: inteiro) **retorna**(lista)
efeito Aloca espaço para um nó da lista. Insere "elemento" no nó alocado. Se não houver espaço suficiente, interrompe a execução. Insere o novo nó no final da lista.

concat(head, tail: lista) **retorna**(lista)
modifica Amplia a primeira lista "head".
efeito Concatena duas listas.

poplast(list: lista, element: inteiro) **retorna**(lista)
modifica element, list.
efeito Retira o último nó da lista "list". Insere o valor do último elemento da lista em "elemento".

firstelt(list: lista, element: inteiro) **retorna**(1)
sinal(lista vazia)
modifica element.
efeito Retira o primeiro valor de uma lista. Insere o valor em "elemento". Retorna zero se a lista estava vazia e um, caso contrário.

push(list: lista, element: inteiro) **retorna**(lista)
sinals(não há espaço de memória suficiente)
modifica list.
efeiito Aloca espaço para um nó da lista.Insere "elemento" no nó alocado. Se não houver espaço suficiente, interrompe a execução. Insere o novo nó no começo da lista.

include(list: lista, element: inteiro) **retorna**(lista)
sinals(não há espaço de memória suficiente)
modifica list.
efeito Se o elemento não pertence à lista, aloca espaço para um nó. Insere "elemento" no nó alocado. Se não houver espaço suficiente, interrompe a execução.

pop(list: lista, element: inteiro) **retorna**(lista)
modifica element, list.
efeito Retira o primeiro nó da lista. Insere o valor em "elemento". Retorna NULL se a lista estava vazia.

belong(list: lista, element: inteiro) **retorna**(1)
sinal(elemento não pertence a lista)
efeito Testa se "element" pertence à lista. Retorna zero se ele não pertence, e um caso contrário.

delelt(list: lista, element: inteiro)
retorna(ponteiro para a lista)
efeito Apaga todas as ocorrências de um elemento da lista.

equal(list1, list2: lista) **retorna**(1)
sinal(diferenças entre as listas)
efeito Verifica a igualdade entre duas listas, em termos da organização dos elementos.

setequal(list1, list2: lista) **retorna**(1)
sinal(diferenças entre as listas)
efeito Verifica se os elementos de list1 são iguais aos elementos de list2.

delete(list: lista) **retorna**(NULL)
modifica list.
efeito Apaga os elementos da lista toda.

merge(list1, list2: lista) **retorna**(lista)
modifica list1.
efeito Insere os elementos de list2 em list1.

inter(list1, list2: lista) **retorna**(lista)
modifica list1.
efeito Insere os elementos que pertencem a list1 e a list2 em uma nova lista, fazendo a intersecção entre as duas listas.

setdif(list1, list2: lista) **retorna**(lista)
modifica list1.
efeito Insere os elementos que pertencem a list1 e não a list2, em uma nova lista, fazendo a diferença entre as duas listas.

copy(list: lista)
retorna(lista)
efeito Copia os elementos de list para uma nova lista. Se não houver memória suficiente, interrompe a execução.

prlist(fbug: FILE, list: lista)
retorna( )
efeito Imprime os elementos de uma lista.

Pilha é alloc_stack, free_stack, push_ stack, pop_stack.

Pilha é uma lista simplemente encadeada onde só o primeiro elemento pode ser retirado e um novo elemento só pode ser inserido na primeira posição da lista.

alloc_stack( )
retorna(pilha)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para um nó da pilha. Se não houver espaço suficiente, interrompe a execução.

free_ stack(stack)
retorna( )
efeito Libera espaço para um nó da pilha.

push_ stack(peak: pilha, pt: inteiro)
retorna(pilha)
sinal(não há espaço de memória suficiente)
modifica peak.
efeito Aloca espaço para um nó da pilha. Insere "pt" no novo nó. Se não houver espaço suficiente, interrompe a execução.

pop_ stack(peak: pilha, pt: inteiro)
retorna(elemento do topo da pilha)
modifica peak.
efeito Remove o nó do topo da pilha. Insere em "pt" o valor do nó retirado.

Processo é checkproc, addprocess, addsource, addsink.

Processo é uma tabela que contém uma relação de todos os processos de alguma forma ligados a uma conexão. Cada linha da tabela corresponde a um processo que contém seu nome, uma lista de índices da tabela de conexões relacionando as conexões que chegam neste processo e uma lista de índices da tabela de conexões relacionando as conexões que saem deste processo.

checkproc(entry: inteiro)
retorna(índice do processo na tabela)
sinal(processo não encontrado)
efeito Verifica se "entry" foi definido como um processo.

addprocess(name: inteiro, source, sink: lista)
retorna(índice da última posição onde foi inserido o processo)
sinal(não há espaço na tabela)
efeito Adiciona um processo à tabela.

addsource(proc, connect: inteiro) **retorna**( )
efeito Adiciona um processo fonte à tabela.

addsink(proc, connect: inteiro) **retorna**( )
efeito Adiciona um processo destino à tabela.

Texto é addtext, gettext, wrtext, getname, wrnitext.

Um texto é uma palavra não reservada. As palavras são inseridas em um vetor de texto através de uma função "hash". Cada palavra é acompanhada do seu comprimento no vetor de texto.

addtext(str: sequência de caracteres, length: inteiro)
retorna(índice "entry" onde foi inserida a palavra)
sinal(não há espaço no vetor)
efeito Calcula a função hash para a palavra. Insere-a na vetor de texto.

gettext(entry: inteiro) **retorna**(palavra)
sinal(texto não encontrado)
efeito Busca o texto.

wrtext(fout: FILE, lvl, entry: inteiro) **retorna**( )
efeito Imprime o texto.

getname(entry: inteiro)
retorna(ponteiro para a palavra)
sinal(texto não encontrado)
efeito Obtém o nome no vetor de texto.

wrnitext (fout: FILE, entry: inteiro) **retorna**( )
efeito Escreve o texto sem identação.

Variável é add_var, mod_var, get_var, get_numvar, get_var_value, get_var_text, belongvar.

Uma variável é um local de armazenamento de valores manipulados por um processo. As variáveis de um processo fazem parte de seu vetor de estados.

add_var(vstate: vetor de estados, varname: inteiro)
retorna( )
efeito Insere "varname" no próximo campo livre do vetor de estados.

mod_var(vstate: vetor de estados, varname, value:inteiro)
retorna( )
sinal(variável não pertence ao processo)
efeito Insere "value" no campo de valor da variável "varname". Se variável não pertence ao processo, interrompe a execução.

get_var(vstate: vetor de estados, varindex: inteiro)
retorna(entrada hash para o nome da variável)
efeito Obtém o índice para o vetor de texto que contém o nome da variável.

get_numvar(vstate: vetor de estados)
retorna(número de variáveis do processo)
efeito Obtém o número de variáveis do processo.

get_var_value(vstate: vetor de estados, varname: inteiro)
retorna(valor inteiro atribuído à variável)
sinal(variável não contém um valor inteiro)
efeito Obtém o valor inteiro de uma variável. Caso ele não contenha um valor inteiro retorna -1.

get_var_text(vstate: vetor de estados, varname: inteiro)
retorna(índice do vetor de texto que contém o valor da variável)
sinal(variável não pertence ao processo)
efeito Obtém o índice do vetor de texto onde se encontra o valor atribuído à variável "varname". A variável deve conter uma palavra ou um caractere.

belongvar(vstate: vetor de estados, varname: inteiro)
retorna(índice do vetor de variáveis)
sinal(variável não pertence ao processo)
efeito Obtém o índice do vetor de variáveis onde se encontra a variável "varname". Retorna o índice desta posição ou um valor vazio caso a variável não seja encontrada neste vetor.

Vetor de estados é create_state, increase_ptext, decrease_ptext.

Vetor de estados é um vetor que contém a abstração pilha e a abstração variáveis.

create_state(numvar: inteiro) **retorna**(vetor de estados)
sinal(não há espaço de memória suficiente)
efeito Aloca espaço para um vetor de estados, isto é, espaço para um ponteiro para uma pilha e um vetor de variáveis com duas vezes "numvar" posições.

increase_ptext(vstate: vetor de estados, pt: inteiro)
retorna(0)
sinal(não há espaço de memória suficiente)
efeito Insere "pt" na pilha do processo e atualiza o ponteiro para o topo da pilha no vetor de estados.

decrease_ptext(vstate: vetor de estados, pt: inteiro)
retorna( )
modifica "pt".
efeito Remove o primeiro elemento da pilha do processo, atribui o valor do elemento à "pt". Caso a pilha esteja vazia, "pt" recebe um valor nulo. Atualiza o ponteiro para o topo da pilha no vetor de estados.

As abstrações de procedimento que não pertencem a uma abstração de dados são descritas a seguir:

computexpr(args: árvore, vstate: vetor de estdos)
retorna(valor resultante da expressão)
sinal(valor não esperado. operador desconhecido)
efeito Calcula o valor de uma expressão booleana ou aritmética. Se o texto de uma variável não for encontrado ou a expressão tiver um operador desconhecido a execução é interrompida.

expressão( ) **returno** (árvore)
sinal (expressão mal formada)
efeito Analisa uma expressão.

execução(tabelas internas do executor-jsd)
retorno( )
sinal(erro durante a execução)
efeito Aloca espaços para a tabela descritora e para as sequencias de dados com multiplicidade. Executa as instruções de cada um dos processos. Implementa o mecanismo de suspende/continua.

executor(arquivo contendo uma especificação JSD)
retorno( )
sinal(erro durante a execução)
efeito Executa uma especificação escrita segundo a linguagem JSD definida aqui.

léxica(uma palavra do arquivo de especificação)
retorno( )
modifica As variáveis globais "token", "string", "strlength".
efeito Obtém uma palavra do arquivo de especificação. Testa se a palavra é uma palavra reservada ou não.

sintaxe(uma declaração do arquivo de especificação)
retorno( )
efeito Obtém uma declaração do arquivo de especificação. Analisa o comando da declaração. Insere as informações sobre os comandos nas devidas tabelas.

CAPÍTULO 6

UM EXEMPLO APLICATIVO

Neste capítulo é apresentado o processo de desenvolvimento de um sistema aplicativo usando as técnicas discutidas nos capítulos precedentes. O sistema aplicativo exemplificado aqui, consiste em automatizar o gerenciamento de uma competição entre os leitores de um jornal, chamado "Daily Racket". Este exemplo cobre alguns pontos do método JSD desde o passo de ações e entidades até a implementação do mesmo através do Executor-JSD.

No método JSD a especificação é desenvolvida, não se parte de uma especificação pronta. O analista é livre para criar e mudar a especificação, junto com o usuário, quantas vezes quiser. No exemplo apresentado neste capítulo, as várias possibilidades de escolhas dentro de cada passo não são levantadas, mas apenas as soluções adotadas. As seções seguintes mostram as soluções em cada passo e o Apêndice C traz alguns relatórios gerados pelo JSD-tool nos passos de um a cinco.

6.1 - A COMPETIÇÃO DO DAILY RACKET

O gerente do Daily Racket planeja uma competição aberta aos leitores que são assinantes. Uma vez que o leitor tornou-se um assinante do jornal, ele pode entrar na competição quantas vezes quiser. Para isso, basta enviar uma ou mais entradas cada vez que o jornal publicar detalhes da competição. Cada entrada deve ser acompanhada por uma gorgeta. A competição é julgada, periodicamente, por um juri de celebridades da televisão e as melhores entradas desde o último julgamento recebem prêmios.

A competição é baseada em um conjunto de três fotografias pequenas e mal impressas de modelos vestindo roupas feias e de mau gosto. Uma entrada consiste de um sequência dessas fotografias em ordem de elegância e bom gosto, junto com um verso expressando um extravagante elogio ao Daily Racket e seu proprietário, Sr. Espertinho de Ville.

No caso de dificuldade no julgamento, é analisado o verso que acompanha a sequência.

Nenhum competidor pode ganhar mais de uma vez. Apenas uma entrada de cada competidor é submetida a julgamento em um sessão do juri. As entradas não premiadas não retornam ao competidor e a gorgeta fica retida no Daily Racket.

Com base nas características da competição, são adotadas as soluções apresentadas nas seções abaixo.

6.2 - AÇÕES E ENTIDADES

A lista de entidades escolhidas, que definirão o escopo do sistema, consta apenas das seguintes entidades:

1) leitor

2) juri

As ações executadas ou sofridas pelas entidades, definidas acima, e os atributos das ações são apresentados abaixo:

1) assinar - tornar-se um freguês regular pagando, anualmente, as cópias do jornal. Atributos: data, nome, endereço;

2) competir - elaborar uma entrada para a competição e enviá-la ao Daily Racket. Atributos: sequência, gorgeta, verso;

3) reunir - encontrar-se uma vez por semana em um lugar pré-fixado, para receber e julgar as entradas. Atributos: data.

4) premiar - premiar uma entrada. Atributos: sequência, verso, tipo-de-prêmio;

As funções desejadas, eventualmente, podem ser extraídas dos próprios atributos. Por exemplo, uma função que pedisse a data em que o leitor tornou-se um assinante, poderia ser extraída do atributo data da ação assinar; outra função que pedisse o total de gorgetas pagas poderia ser extraída do atributo gorgeta da ação competir, daí a importância da definição dos atributos em cada ação.

6.3 - ESTRUTURA DAS ENTIDADES

Neste passo as ações de cada entidade são colocadas em ordem de ocorrência. A estrutura da entidade leitor é mostrada na Figura 6.1 e a estrutura da entidade juri está ilustrada na Figura 6.2.

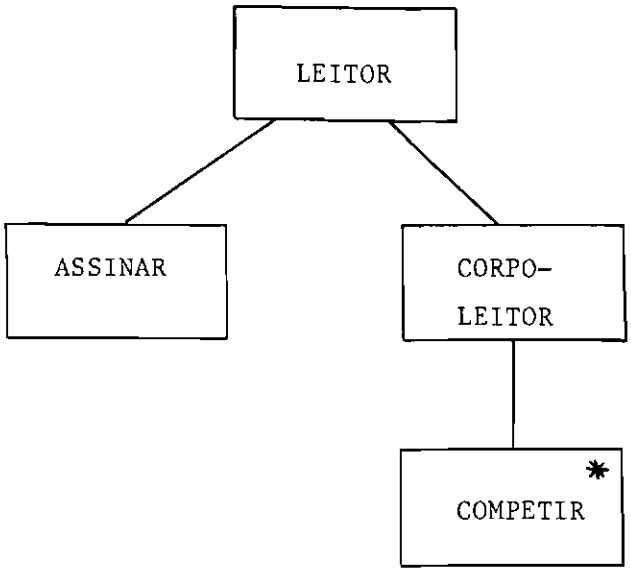


Fig. 6.1 - Estrutura da entidade leitor.

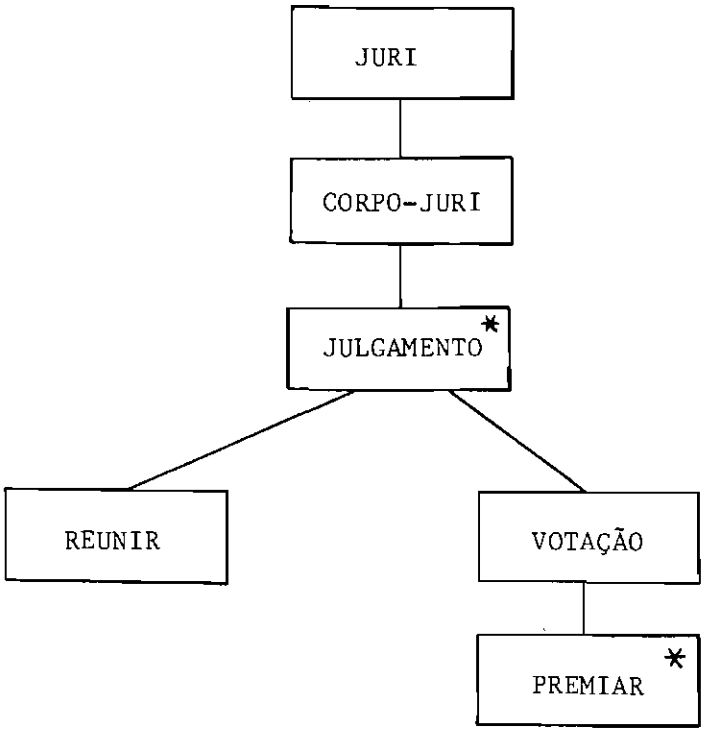


Fig. 6.2 - Estrutura da entidade juri.

6.4 - MODELO INICIAL

No passo 3 são criados os processos modelos que vão simular, no computador, o comportamento das entidades especificadas. Além dos processos, são definidas as conexões entre eles e o mundo real.

No sistema Daily Racket são criados os processos modelos LEITOR-1 e JURI-1. A entidade leitor, LEITOR-0, comunica-se com o processo modelo LEITOR-1 através da troca de mensagens, cada vez que ela sofre ou executa uma ação . Da mesma forma, a entidade juri, JURI-0, comunica-se com o processo modelo JURI-1 através de troca de mensagens.

O Diagrama de Especificação do Sistema definido para a competição Daily Racket, no passo do modelo inicial, é mostrado na Figura 6.3.

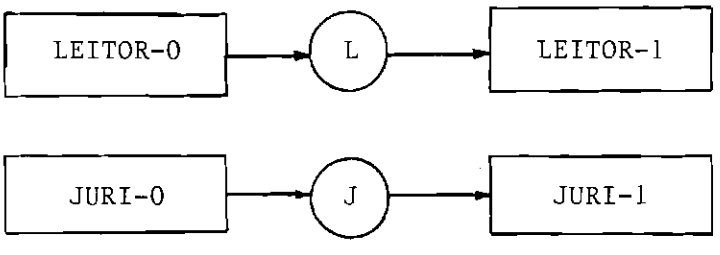


Fig. 6.3 - Diagrama de Especificação do Sistema Daily Racket, no passo do modelo inicial.

Na apresentação do texto de estrutura dos processos modelos são inseridos os comandos necessários para realizar a comunicação entre eles e o mundo real.

O juri não pode ficar reunido eternamente, como está mostrado no diagrama de estrutura da entidade juri, na Seção 6.3. Assim, após reunir, fazer as devidas premiações a reunião é desfeita e os jurados dispersam. Esta ação é inserida, então, no texto de estrutura do processo modelo JURI-1.

A estrutura textual dos processos modelos LEITOR-1 JURI-1 são apresentadas a seguir:

```
LEITOR-1 seq
         read L;
         assinar;
         read L;
         CORPO-LEITOR iter
                      competir;
                      read L;
         CORPO-LEITOR end
LEITOR-1 end


JURI-1 seq
       read J;
       CORPO-JURI iter
                  JULGAMENTO seq
                             reunir;
                             read J;
                             VOTAÇÃO iter (prêmio)
                                     premiar;
                                     read J;
                             VOTAÇÃO end
                             dispersar;
                  JULGAMENTO end
       CORPO-JURI end
JURI-1 end
```

Os processos modelos LEITOR-1 e JURI-1 podem ser implementados e modelarem o comportamento do mundo real dentro do computador, em termos das entidades definidas. Neste ponto entram em cena as funções!

6.5 - FUNÇÕES E SINCRONIZAÇÃO DO SISTEMA

Baseado no modelo do mundo real são inseridas as funções solicitadas pelo comprador ou usuário do sistema.

A lista de funções especificadas pelo Sr. Espertinho de Ville diz que o sistema deve:

1) ser capaz de mostrar dados sobre o leitor, tais como, nome e endereço;

2) armazenar as informações (nome e endereço) sobre o leitor;

3) elaborar um relatório semanal para que o proprietário, Sr. Espertinho de Ville, possa acompanhar a competição. Este relatório deve ter o seguinte formato e conteúdo :

   Semana nnn - Relatório de Entradas
   Entradas da Semana - eeee
   Total de Entradas - ttttt

4) informar, sempre que pedido, o número de entradas feitas por cada um dos leitores;

5) elaborar uma lista de entradas para ser julgada pelo juri a cada reunião, assegurando que não existe mais de uma entrada de um mesmo leitor e nenhuma entrada que já ganhou um prêmio;

6) elaborar um relatório de atividades do juri, mostrando se foi premado um leitor que não tinha uma entrada na lista. (Sr. Espertinho de Ville é um tanto desconfiado);

7) informar o número total de prêmios distribuídos até a data presente, quando solicitado.

As funções 1 e 2 são, claramente, funções embutidas, pois o processo leitor já guarda estas informações em suas variáveis. A função 2 está pronta no processo modelo LEITOR-1. Mas, para realizar a função 1, os dados sobre o leitor devem ser exteriorizados, bastando, para isso, inserir um comando de escrita neste processo para que ele mostre as informações sobre o leitor.

A função 3, que computa o número de entradas enviadas em uma semana, não pode ser uma função embutida simples, pois o processo modelo só reproduz o comportamento de um leitor. Assim, ela é executada pelo processo função COMPENTS, conectado ao processo modelo LEITOR-1 através de uma sequência de dados.

Para exibir o número de entradas feitas por um leitor até o presente momento, é criado o processo função NUMENTS. Onde o usuário entra com a identificação do leitor e este processo retorna o número de entradas feitas pelo leitor. Esta é, portanto, uma função imposta simples. O processo LEITOR-1 é munido de um contador e de operações para atualizar o número de entradas. O processo NUMENTS conecta-se ao processo LEITOR-1 através de um vetor de estados para obter o número de entradas computadas por este processo. E por uma sequência de dados, liga-se ao mundo real para que o gerente possa pedir o número de entradas feitas por um certo leitor.

A função 5 dá origem a um outro processo função que liga o processo LEITOR-1 ao processo JURI-1, através de algumas sequências de dados.

Uma verificação se o ganhador faz parte da lista de entrada, o que é pedido na função 6, é desnecessário no sistema automatizado, pois o juri não interfere na lista de nomes, ele simplesmente aperta uma tecla no terminal dizendo se o competidor deve ganhar ou não o prêmio.

A função 7, também é uma função imposta simples, realizada pelo processo NUMPREMIOS, o qual está conectado ao mundo externo via uma sequência de dados e ao processo modelo JURI-1 através de um vetor de estados. O processo JURI-1 é quem mantém um variável atualizada contendo o número total de prêmios distribuídos.

O diagrama de especificação do sistema que envolve todas estas funções (processos) é mostrado na Figura 6.4 e a estrutura textual de cada um dos processos é apresentada em seguida.

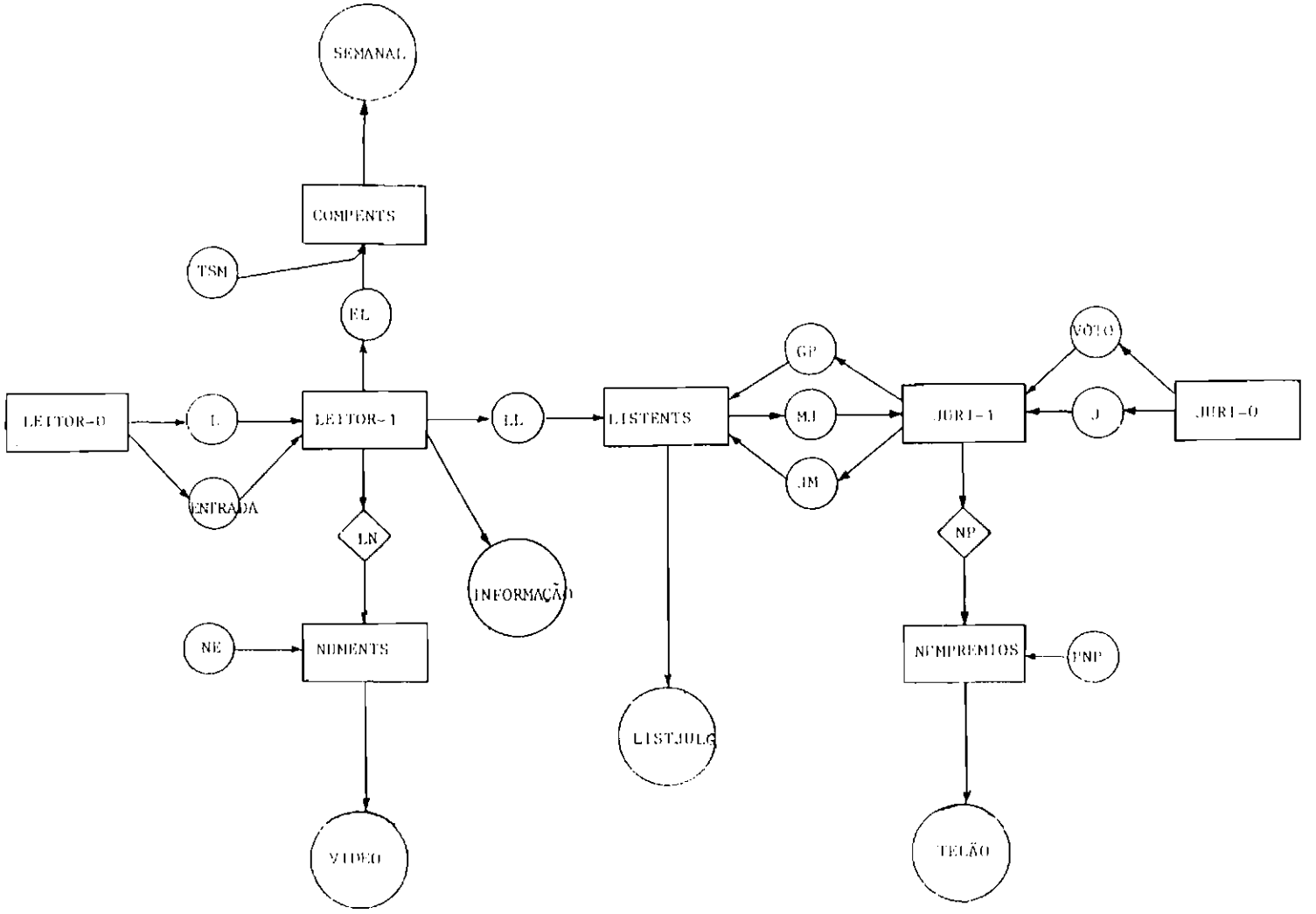


Fig. 6.4 - Diagrama de Especificação do Sistema Daily Racket, no passo de função.

A descrição dos processos especificados é:

```
% Processo LEITOR-1

      LEITOR-1 seq
               read L;
               write L para INFORMAÇÃO
               read ENTRADA;
               ce = 0;
               COMPETIR iter (ainda há entradas)
                        write ENTRADA para LL;
                        write 1: EL,
                        ce = ce + 1;
                        write L para INFORMAÇÃO;
                        read ENTRADA;
               COMPETIR end
      LEITOR-1 end

% Processo JURI-1

      JURI-1 seq
             read J;
             npremios = 0;
             JULGAMENTO iter (há encontro)
                        write J para JM;
                        read MJ;
                        VOTAÇÃO iter (há entradas)
                                write Julgue a sequência abaixo
                                                    para VÍDEO;
                                read VOTO;
                                write VOTO para GP;
                                CONTPREMIO sel (ganhou)
                                           nprêmios = npremios+1;
```

```
                        CONTPREMIO alt (não ganhou)
                        CONTPREMIO end
                        read MJ;
                VOTAÇÃO end
                DISPERSAR seq
                          write Fim da sessão para VÍDEO;
                DISPERSAR end
                read J;
        JULGAMENTO end
JURI-1 end

% Processo COMPENTS

COMPENTS seq
        total = 0;
        estasm = 0;
        nsemana = 0;
        SEMANAS iter (ainda há competição)
                read TSM&EL;
                CONTAGEM sel (não fim da semana)
                        estasm = estasm + 1;
                CONTAGEM alt (fim da semana)
                        nsemana = nsemana + 1;
                        total = total + estasm;
                        FIMSEMANA seq
                                  write Relatório de En-
                                     tradas para SEMANAL;
                                  write Semana nsemana
                                           para SEMANAL;
                                  write Entradas da sema-
                                  na estasm para SEMANAL;
                                  write Total de Entradas
                                      total para SEMANAL;
                        FIMSEMANA end
```

```
                                estasm = 0;
                        CONTAGEM end
                SEMANAS end
        COMPENTS end

% Processo NUMENTS

        NUMENTS seq
                read NE;
                NENTRS iter (é nome de um leitor)
                        getsv LN;
                        write nome, endereço e  número de entradas
                                                do leitor para VÍDEO;
                        read NE;
                NENTRS end
        NUMENTS end

% Processo LISTENTS

        LISTENTS seq
                 nenc = 0;
                 read JM;
                 SEQUÊNCIAS iter (há sequências)
                           LEITORES iter ( há leitores inscritos)
                                    read LL;
                                    write LL para LISTJULG;
                                    write LL para MJ;
                                    read GP;
                           LEITORES end
```

```
                    ENCONJURI seq
                              nenc = nenc + 1;
                              write Fim do julgamento de
                                 número nenc para LISTJULG;
                              write sequência final para
                                                    MJ;
                    ENCONJURI end
                    read JM;
          SEQUÊNCIAS end
LISTENTS end

% Processo NUMPREMIOS

NUMPREMIOS seq
           read PREMIOS;
           getsv NP;
           write Número de prêmios distribuídos
                        igual a nprm para TELÃO;
NUMPRÊMIOS end
```

A sincronização consiste da verificação de todas as conexões definidas no sistema, bem como de uma checagem no texto de cada processo envolvido na conexão. Se existe uma conexão entre dois processos, o texto do processo fonte da conexão deve conter comandos de escrita e o texto do processo destino deve conter comandos de leitura ou obtenção dos valores da conexão. Esta verificação deve ser feita logo após a definição dos processos, ou seja, após a fase de especificação das funções necessárias. Além disso, neste passo, são levantadas as restrições temporais (tempos de resposta, frequências de consulta, etc) que precisam ser satisfeitas na implementação. Estas informações seriam descritas informalmente como uma lista de restrições.

6.6 - IMPLEMENTAÇÃO

A implementação será feita utilizando o Executor-JSD, portanto poucos ajustes serão necessários na especificação, para que ela possa ser executada.

O número de processadores disponíveis é apenas um. A especificação, ou seja, a descrição sequencial dos processos e o diagrama de especificação do sistema devem ser transcritos para a linguagem de especificação JSD definida aqui, para que esta especificação possa ser executada como está.

O Executor-JSD implementa um mecanismo de multiprogramação, assim os processos implementados correspondem aos mesmos processos especificados, a menos de detalhes particulares, como:

1) o nome da conexão entre o processo modelo JURI-1 e o mundo real, foi trocado de J para ENCTRO por ficar mais simpático na tela de interface entre o Executor-JSD e o usuário;

2) o sufixo do nome dos processos modelos LEITOR-1 e JURI-1 foi abolido, por não haver outros níveis dos mesmos processos;

3) a sequência de entradas do leitor, isto é, a sequência de fotografias é implementada como uma sequência de três caracteres;

4) são definidas as variáveis necessárias e o número de instâncias de cada processo;

5) o processo LEITOR foi especificado com apenas três instâncias para facilitar a visualização da execução;

6) deve ser feita ainda, uma adaptação da declaração dos comandos à sintaxe da linguagem de especificação executável.

A especificação pronta para ser executada é apresentada a seguir de acordo com sintaxe apresentada no Apêndice B. Esta especificação foi criada pelo editor de texto do turboc e está contida em um arquivo denominado "competic.jsd".

```
spec  LEITOR: 3 ( ce, nome, ender, sequencia, fiment, flag ),
      COMPENTS : 1 ( entr, nsemana, estasm, total ),
      NUMENTS : 1 ( nentradas, nome, ender, nleitor ),
      JURI: 1  ( npremios, juri, sequencia, goup, neg, j, final,
           ganhou ),
      LISTENTS: 1 ( sequencia, ganhou, i, j, n, enc ),
      NUMPREMIOS: 1 ( nprm, pedido, sim );

%   Processo   LEITOR

seq LEITOR :
       read L to ( nome ),
       read L to ( ender ),
       write ( nome ) to VIDEO,
       write ( ender ) to VIDEO,
       read ENTRADA to (sequencia),
       ce = 0,
       flag = 1,
       fiment = fim,
       competir;
       iter competir : ( sequencia != fiment )
              write (sequencia ) to LL,
              write ( flag ) to EL,
              ce = ce + 1,
              write ( nome ) to VIDEO,
              write ( ender ) to VIDEO,
```

```
            read ENTRADA to (sequencia);
end LEITOR;

%   Processo  COMPENTS

seq COMPENTS :
      total = 0,
      estasm = 0,
      nsemana = 0,
      semanas;
      iter semanas : ( nsemana < 20 )
            read TSM&EL to (entr),
            contagem;
            sel contagem : ( entr == 1 )
                  estasm = estasm + 1
            alt ( entr == 0 )
                  nsemana = nsemana + 1,
                  total = total + estasm,
                  fimsemana,
                  estasm = 0;
                  seq fimsemana :
                        write Relatorio de Entradas to SEMANAL,
                        write .. Semana .. to SEMANAL,
                        write ( nsemana ) to SEMANAL,
                        write Entradas da Semana  to SEMANAL,
                        write ( estasm ) to SEMANAL,
                        write Total de Entradas to SEMANAL,
                        write ( total ) to SEMANAL;
end COMPENTS;
```

```
%   Processo  NUMENTS

seq NUMENTS :
      read NE to ( nleitor ),
      numents;
      iter numents : ( nleitor < 3 )
            getsv LN( nleitor) to ( nentradas, nome, ender ),
            write ( nome ) to VIDEO,
            write ( ender ) to VIDEO,
            write ( nentradas ) to VIDEO,
            read NE to ( nleitor );
end NUMENTS;

%   Processo JURI

seq JURI:
      neg = n,
      final = fim,
      ganhou = g,
      npremios = 0,
      read ENCTRO to ( j ),
      julgamento;
      iter julgamento : ( j != neg )
            write ( j ) to JM,
            read MJ to ( sequencia ),
            votacao,
            read ENCTRO to ( j );
            iter votacao : ( sequencia != final )
                  write Julgue a sequencia to VIDEO,
                  write ( sequencia ) to VIDEO,
                  read VOTO to ( goup ),
                  write ( goup ) to GP,
                  contpremio,
                  read MJ to ( sequencia );
```

```
                    sel contpremio: (goup == ganhou)
                          npremios = npremios + 1
                    alt (goup != ganhou);
end JURI;

%   Processo  LISTENTS

seq LISTENTS :
      n = 0,
      enc = e,
      read JM to ( j ),
      sequencias;
      iter sequencias : ( j == enc )
            i = 0,
            leitores,
            enconjuri,
            read JM to ( j );
            iter leitores : ( i < 3 )
                  read LL(i) to ( sequencia ),
                  write Sequencias to LISTJULG,
                  write (sequencia ) to LISTJULG,
                  write ( sequencia ) to MJ,
                  i = i + 1,
                  read GP to ( ganhou );
      seq enconjuri :
            n = n + 1,
            write Fim do Julgamento de Numero to LISTJULG,
            write ( n ) to LISTJULG,
            sequencia = fim,
            write (sequencia) to MJ;
end LISTENTS;
```

```
% Processo NUMPREMIOS

seq NUMPREMIOS :
      sim = s,
      read PREMIOS to (pedido),
      pegnumpr;
      iter pegnumpr: ( pedido == sim )
            getsv NP to (nprm),
            write Numero de Premios Distribuidos to VIDEO,
            write (nprm) to VIDEO,
            read PREMIOS to (pedido);
end NUMPREMIOS;
%
%  Definição das conexões
%
external stream L --> LEITOR;
external stream ENTRADA --> LEITOR;
external stream NE --> NUMENTS;
external stream ENCTRO --> JURI;
external stream VOTO --> JURI;
external stream PREMIOS --> NUMPREMIOS;
stream EL : many LEITOR --> COMPENTS;
stream LL : many LEITOR --> LISTENTS;
stream MJ : LISTENTS --> JURI;
stream GP : JURI --> LISTENTS;
stream JM : JURI --> LISTENTS;
state LN : many LEITOR --> NUMENTS;
state NP : JURI --> NUMPREMIOS;
report LEITOR --> VIDEO;
report COMPENTS --> SEMANAL;
report LISTENTS --> LISTJULG;
time TSM --> COMPENTS;
rough merge TSM, EL;
```

6.7 - EXECUÇÃO

Ao executar uma especificação,o Executor-JSD gera dois arquivos intermediários: um arquivo de erros de sintaxe e um arquivo de depuração, o qual foi útil durante o desenvolvimento do mesmo e pode servir para a manutenção do sistema ou a um usuário mais curioso.

O arquivo de erros de sintaxe contém as declarações da especificação numeradas na ordem em que elas aparecem no arquivo fonte. Uma indicação de erro de sintaxe é marcada, neste arquivo, com um asterisco logo após a detecção do erro, seguido de uma mensagem explicativa. Este arquivo recebe o nome do arquivo fonte seguido da extensão ".err".

Existem alguns tipos de erros não detectados pelo Executor-JSD, como por exemplo, um erro em uma palavra reservada como **to**, **write**, ou mesmo a ausência de operadores em uma expressão, ou a ausência do sinal de atribuição, e outros. Cabe ao analista o cuidado ao escrever a especificação para evitar estes erros, pois apesar dos erros de sintaxe, a especificação será executada. Dependendo da gravidade do erro ela pode ser executada até o final ou deverá ser abortada externamente.

Para ilustrar o arquivo de erros de sintaxe gerado pelo Executor-JSD, para o exemplo acima, foram introduzidos alguns erros no arquivo fonte da especificação. O conteúdo das duas primeiras páginas da listagem do arquivo "competic.err" gerado, é mostrado a seguir.

```
Executor-JSD 1.1 Wed Apr 27 18:44:23 1988  competic.err
                                                        Pagina 1
0001: spec LEITOR: 3 (ce, nome, ender, sequencia, fiment, flag ),
0002:      COMPENTS : 1 ( entr, nsemana, estasm, total ),
0003:      NUMENTS : 1 ( nentradas, nome, ender, nleitor ),
0004:      JURI : 1 ( npremios, juri,  sequencia,  goup,  neg, j,
                  final, ganhou ),
0005:     LISTENTS 1 ( sequencia, ganhou, i, j, n, enc ),
                     *
*** erro: ':' esperado
0006:     NUMPREMIOS: 1 ( nprm, pedido, sim );
0007:
0008:    Processo   LEITOR
0009:
0010: seq LEITOR :
          *
*** erro: unknown command
0011:     read L to ( nome ),
0012:     read L to ( ender ),
0013:     write ( nome ) to VIDEO,
0014:     write ( ender ) to VIDEO,
0015:     rea ENTRADA to (sequencia),
0016:     ce = 0,
0017:     flag = 1,
0018:     fiment = fim,
0019:     competir;
0020:
0021:     iter competir : ( sequencia != fiment )
0022:          write (sequencia ) to LL,
0023:          write ( flag ) to EL,
0024:          ce = ce + 1,
```

```
0025:          write ( nome ) to VIDEO,
0026:          write ( ender ) to VIDEO,
0027:          read ENTRADA to (sequencia)
0028: end LEITOR;
          *
*** erro: ';' esperado
0029:
0030: %   Processo  COMPENTS
0031:
0032: seq COMPENTS
0033:     total = 0,
               *
*** erro: ':' esperado
0034:     estasm = 0,
0035:     nsemana = 0,
0036:     semanas;
0037:
0038:     iter semanas : ( nsemana < 20 )
0039:          read TSM&EL to (entr),
0040:          contagem;
0041:
0042:          sel contagem : ( entr == 1 )
0043:               estasm = estasm + 1
0044:          alt ( entr == 0 )
0045:               nsemana = nsemana + 1,
0046:               total = total + estasm,
0047:               fimsemana,
0048:               estasm = 0;
0049:
0050:               seq fimsemana :
```

```
Executor-JSD 1.1 Wed Apr 27 18:44:23 1988  competic.err
                                                        Pagina 2
0051:                   write Relatorio de Entradas to SEMANAL,
0052:                   write .. Semana .. to SEMANAL,
0053:                   write ( nsemana ) to SEMANAL,
0054:                   write Entradas da Semana  to SEMANAL,
0055:                   write ( estasm ) to SEMANAL,
0056:                   write Total de Entradas to SEMANAL,
0057:                   write ( total ) to SEMANAL;
0058: end COMPENTS;
0059:
0060: %   Processo  NUMENTS
0061:
0062: seq NUMENTS :
0063:     read NE to ( nleitor ),
0064:     numents;
0065:
0066:     iter numents : ( nleitor ( 3 )
0067:          getsv LN( nleitor) to ( nentradas, nome, ender ),
0068:          write ( nome ) to VIDEO,
```

O segundo arquivo gerado pelo Executor-JSD, o arquivo de depuração, não é de interesse do usuário para a manutenção do sistema. Este arquivo mostra o conteúdo de todas as estruturas internas (ou **objetos**, como foram chamadas na descrição de projeto apresentada no Capítulo 5) geradas após a análise sintática da especificação. A execução propriamente dita, da especificação, se dá com base no conteúdo de tais estruturas.

Ao executar o sistema da competição Daily Racket, as entidades devem interagir com o sistema respondendo às perguntas que vão sendo feitas pelo sistema.

Os primeiros processos executados são as instâncias de LEITOR. A interação com o usuário é como mostrado abaixo, onde o texto grifado indica a resposta da entidade e o não grifado são as saídas do sistema.

```
A)jsc
Interpretador-JSD as suas ordens.

De o nome do arquivo de especificacao: compatic.jsc)

De o valor de _ --> nome  [LEITOR-i0]: Ana Maria Amorosio

De o valor de L --> ender  [LEITOR-i0]: Rua Paraiouna,55

nome= Ana Maria Amorosio,
ender= Rua Paraiouna,55,
De o valor de ENTRADA --> sequencia  [LEITOR-i0]: qwe

nome= Ana Maria Amorosio,
ender= Rua Paraiouna,55,
De o valor de ENTRADA --> sequencia  [LEITOR-i0]: -

De o valor de L --> nome  [LEITOR-i1]: Homero Ribeiro

De o valor de L --> ender  [LEITOR-i1]: Pca Conego Lima,92

nome= Homero Ribeiro,
ender= Pca Conego Lima,92,
De o valor de ENTRADA --> sequencia  [LEITOR-i1]: sdf

nome= Homero Ribeiro,
ender= Pca Conego Lima,92,
De o valor de ENTRADA --> sequencia  [LEITOR-i1]: -

De o valor de L --> nome  [LEITOR-i2]: Mario C.P. Almeida

De o valor de L --> ender  [LEITOR-i2]: Rua Dr. Quirino,725

nome= Mario C.P. Almeida,
ender= Rua Dr. Quirino,725,
De o valor de ENTRADA --> sequencia  [LEITOR-i2]: ghj

nome= Mario C.P. Almeida,
ender= Rua Dr. Quirino,725,
De o valor de ENTRADA --> sequencia  [LEITOR-i2]: -
```

O encerramento de uma semana é indicado através de um valor "zero" enviado a TSM. O relatório SEMANAL é emitido, segundo o formato mostrado a seguir, relativa a primeira semana da competição.

```
De o valor de TSM  [COMPENTS-10]: 0

Relato SEMANAL --> Relatorio de Entradas
Relato SEMANAL --> .. Semana ..
nsemana=
Relato SEMANAL --> Entradas da Semana
estasm=     3
Relato SEMANAL --> Total de Entradas
total=     3
De o valor de TSM  [COMPENTS-10]: -
```

Quando Sr. Espertinho deseja saber o número de entradas feitas por um certo leitor, ele envia o número do leitor (um valor entre 0 e 2) à conexão NE. A saída do sistema é como apresentado abaixo.

```
De o valor de NE --> nleitor  [NUMENTS-10]: 0

nome= Ana Maria Ambrosio,
ender= Rua Paraibuna,55,
nentradas=
De o valor de NE --> nleitor  [NUMENTS-10]: 1

nome= Homero Ribeiro,
ender= Pca Conego Lima,92,
nentradas=     1,
De o valor de NE --> nleitor  [NUMENTS-10]: 2

nome= Mario C.P. Almeida,
ender= Rua Dr. Quirino,725,
nentradas=     1,
```

O relatório de todas as sequências emitidas a julgamento, a cada encontro, estão no relatório LISTJULG.

```
Relate LISTJULG -->   Sequencias
sequencia= awe,

Relate LISTJULG -->   Sequencias
sequencia= sdf,

Relate LISTJULG -->   Sequencias
sequencia= ghj,

Relate LISTJULG -->   Fim do Julgamento de Numero
n=     1,
```

A entidade juri deve avisar o sistema cada vez que há uma nova reunião. Isto é feito com uma mensagem "e" enviada à conexão ENCTRO, como ilustrado abaixo.

```
De o valor de ENCTRO --> j  [JURI-10]: e
```

No julgamento autorizado, basta um dos jurados emitir uma mensagem "p" ou "g" se a sequência deve perder ou ganhar, após um consenso do juri.

```
Relate VIDEO -->   Julgue a sequencia
sequencia= awe,
De o valor de VOTO --> goud  [JURI-10]: p
```

```
Relate VIDEO --> Julgue a sequencia
sequencia= sdf,
De o valor de VOTO --> goup [JURI-10]: p

Relate VIDEO --> Julgue a sequencia
sequencia= gnj,
De o valor de VOTO --> goup [JURI-10]: g
```

Para saber o número de premios já distribuídos, Sr. Espertinho deve enviar à conexão PREMIOS, a mensagem "s".

```
De o valor de PREMIOS --> pedido [NUMPREMIOS-10]: s

Relate VIDEO --> Numero de Premios Distribuidos
norm= 1.
De o valor de PREMIOS --> pedido [NUMPREMIOS-10]: -
```

A sequência de saídas é apresentada ao usuário, de forma intercalada de acordo com a ordem de execução dos vários processos. Logo, a apresentação de todos os relatórios mais as perguntas às entidades, em uma única tela, não se revela muito simpática aos olhos do usuário.

A organização das saídas do sistema apresentada acima, isto é, dividida por áreas de interesse das várias entidades, poderia ser uma opção de saída se houvessem vários processadores ou simplesmente vários terminais de vídeo ligados ao único processador.

CAPÍTULO 7

CONCLUSÃO

As técnicas de especificação JSD são claramente compreensíveis e independentes da implementação. No método JSD, a decomposição do trabalho de desenvolvimento do sistema não é confundida com a decomposição do problema. Este método separa bem os aspectos de modelo dos aspectos de funções: nos passos iniciais as preocupações são associadas ao modelo da realidade e somente nos passos subsequentes são levantadas as funções do sistema; separa também, os aspectos de projeto dos aspectos de implementação e ainda, os aspectos dinâmicos dos aspectos estáticos do sistema a ser desenvolvido.

Na fase de transformação, o analista não fica perdido pois, o JSD fornece algumas técnicas para transformar o modelo do sistema em um processamento eficiente. As transformações sugeridas pelo Jackson são, a inversão, o desmembramento de programas e a separação de vetor de estados.

Deve ser lembrado que, existem aspectos de especificação de requisitos para os quais um modelo operacional possui pouco ou nenhum auxílio. Estes, incluem requisitos relacionados com o gerenciamento do desenvolvimento do sistema, tais como, prazo, custos, pessoal e recursos disponíveis, etc.

Entretanto, com relação aos aspectos organizacionais, no método JSD podem ser estabelecidos os documentos que devem ser gerados no final de cada passo, bem como, os marcos para o encerramento de cada fase. Ou pode-se manter apenas a especificação final, a qual é executável, de acordo com o objetivo dos modelos operacionais.

Além destas vantagens, (ou por causa destas vantagens) o método JSD é bastante difundido.

A construção do Executor-JSD contribuiu significativamente para o entendimento dos problemas, vantagens e desvantagens de se executar diretamente uma especificação JSD, e proporcionou uma demonstração automática da interface com o usuário.

A organização e formalização dos requisitos de um sistema desenvolvido através de um modelo operacional parece ser viável e não absurdamente complexa.

O método JSD não apresenta dificuldade ao usuário e o desenvolvimento de um sistema fica muito mais simpático ao usuário com o auxílio do JSD-tool. Desta forma, tornar a especificação um documento escrito em linguagem formal (para que possa ser executada) é uma tarefa trivial, que foi suavizada durante os cinco primeiros passos do método.

O JSD-tool, uma ferramenta que auxilia o analista a desenvolver um sistema segundo o método JSD contém alguns cheques de consistência referentes aos passos de um a quatro. Outros cheques relativos à fase de implementação podem ser construidos. Um tipo de controle sobre o curso de execução também poderia ser construído para analisar e avaliar o tempo de execução de partes críticas do sistema, as quais poderiam ser alteradas até atingirem o tempo de execução desejado.

A definição dos dados na linguagem de especificação JSD está bastante precária sob o ponto de vista da implementação (não é permitido ao usuário definir tipos de dados). Porém, sob o ponto de vista da especificação este fato não deve ser uma preocupação do usuário na fase de especificação, então, por este lado, a impossibilidade de definição de tipos de dados causa maior conforto ao responsável pela especificação.

Outro conforto ao usuário, parece ser o pequeno número de tipos de comandos disponíveis na linguagem de especificação definida aqui (segundo o método JSD), bem como, a simplicidade da sintaxe dos mesmos.

Ainda com relação aos comandos, uma sugestão que poderia ser feita, baseada no conhecimento de sistemas concorrentes e de tempo real e em metodologias voltadas para esta área da computação, é a introdução de comandos, na linguagem de especificação, tais como, suspender, ativar, alterar a prioridade de um processo. Tais comandos poderiam ser introduzidos no passo de sincronização onde são apenas levantadas as restrições de tempo do sistema. Eles poderiam permitir ao usuário resolver os problemas referentes às restrições de tempo e sincronização do sistema sem alterar a metodologia JSD.

Como dito anteriormente, este trabalho é uma ferramenta experimental, e portanto, possui uma série de limitações e simplificações. Por outro lado, ele atingiu seu objetivo de demonstrar a viabilidade do desenvolvimento de sistemas capazes de facilitar imensamente o trabalho de desenvolvimento e a manutenção de sistemas de software através da execução da própria especificação.

REFERÊNCIAS BIBLIOGRÁFICAS

BALZER, R.; GOLDMAN, N. Principles of good software specification and their implications for specification language. In: SPECIFICATION OF RELIABLE SOFTWARE CONFERENCE, Boston, MA, Apr. 1979. Proceedings. New York,NY, IEEE Computer Society, 1979, p. 58-67.

BAUER, F.L. Software engineering. In: GOOS, G.; HARTMANIS, J., ed. Software engineering - an advanced course. Berlin, Springer-Verlag, 1975. p. 522-545.

CAMERON, R.J. An overview of JSD. IEEE Transaction on Software Engineering, SE-12(2):222-240, Feb. 1986.

DEARNLEY, P.A.; MAYHEW, P.J. In favour of system prototypes and their integration into the systems development cycle. The Computer Journal, 26(1):36-42, 1983.

FAIRLEY, R. Planning a software project. In: _____ Software engineering concepts. New York NY, McGraw-Hill, 1985. cap. 2, p. 30-63.

HANSEN, B.P Concurrent process.In: Operating system principles. Englewood Cliffs, NJ, Prentice-Hall, 1973. cap. 3, p. 55-131.

HOWLEY, P.P.JR. A comprehensive software testing methodology. In: SOFTWARE ENGINEERING STANDARDS APLICATION WORKSHOP, 2., San Francisco, CA, 1983. Annals. Silver Spring, MD, IEEE Computer Society, 1983, p. 156-163.

INSTITUTE OF ELETRICAL AND ELECTRONICS ENGINEERS (IEEE). IEEE guide to software requirements specifications, New York, NY, 1984. (ANSI/IEEE STd 830-1984).

JACKSON, M.A. System development. Englewood Cliffs, NJ, Prentice-Hall, 1983.

KERNIGHAN, B.W.; RITCHIE, D.M. The C programming language. Englewood Cliffs, NJ, Prentice-Hall, 1978.

LISKOV, B. Modular program construction using abstractions. Cambridge, MA, Laboratory for Computer Science. Massachusetts Institute of Technology, Sept., 1979. p. 2-43.(Computation Structures Group Memo 184).

URBAN, S.D.; URBAN, J.E.; DOMINICK, W.D. Using an executable specification language for an information system. IEEE Transaction on Software Engineering, SE-11(7):598-605, July 1985.

VELASCO, F.R.D. JSD-Tool reference manual. Tyngsboro, MA. Wang Institute Graduate Studies, 1986a. (TR-86-09).

——— JSD-Tool user manual. Tyngsboro, MA. Wang Institute Graduate Studies, 1986b. (TR-86-08).

YEH, R.T.; ZAVE, P.; CONN, A.P.; COLE, G.E.JR. Software requirements: new directions and perspectives. In: VICK, C.R.; Ramamoorthy, C.V. Handbook of software engeneering. New York, NY, Van Nostrand Reinhold, 1984. p. 519-543.

ZAVE, P. An operational approach to requirements specification for embedded systems. IEEE Transactionon Software Engineering, SE-8(3):250-269, May 1982.

ZAVE, P. Operational specification languages. In: ACM ANNUAL CONFERENCE, New York, NY, Oct. 24-26, 1983. Proceedings, 38. New York, NY, ACM, 1983, p. 1-9.

——— The operational versus the conventional approach to software development. Comunications of ACM, 27(2):104-118, Feb. 1984.

——— PAISLey user documentation. In: Tutorial. Murray Hill, NJ AT&T Bell Laboratories, 1987. v.2, p. 1-3. (Relatório interno da AT&T Bell)

APÊNDICE A

GLOSSÁRIO DE TERMOS

abstração - descrição de uma realidade, omitindo, propositalmente, certos aspectos;

ação - um evento no qual uma ou mais entidades participam ou sofrem;

analistas - as pessoas que realizam o produto para os compradores;

árvore - um tipo de estrutura usada extensivamente no JSD para dados e programas;

backtracking - técnica para manipulação de falhas de informação para uma decisão de tempo real. Faz-se uma suposição e se ela for comprovadamente errada, basta abandoná-la;

canal - um recurso de implementação no qual um processo é invertido com respeito a duas ou mais sequências de dados simultaneamente;

compradores - as pessoas que pagam pelo produto e geralmente (não necessáriamente) decidem os requisitos;

DES = Diagrama de Especificação do Sistema - diagrama mostrando os processos do sistema e as conexões entre eles;

desmembramento - uma transformação na qual o texto ou o vetor de estados de um processo ou a sequência de dados é quebrada em um número de partes conveniente e eficiente para execução;

diagrama de estrutura - um diagrama que representa a árvore de estrutura de uma sequência de dados ou um texto de processo;

DIS = Diagrama de Implementação do Sistema - diagrama mostrando como um sistema é implementado;

entidade - um objeto do mundo real que participa de um conjunto de ações ordenadas no tempo;

entidade marsupial - uma entidade que nasce da estrutura de outra;

escalonamento - um interrelacionamento de execuções de dois ou mais processos, tal que elas possam ser tratadas como uma execução de um único processo e realizada em um único processador;

estrutura textual - uma representação textual da estrutura de árvore de um processo ou de uma sequência de dados, mostrando as condições sobre os componentes de iteração e seleção;

função - uma ação ou um conjunto de ações executadas pelo sistema e cujo resultado são as saídas desejadas;

função de interação - uma função que interage com o processo modelo escrevendo buffers a sequências de dados que o processo possa ler e assim alterar seu estado;

função embutida - função cujas operações de saída requeridas são inseridas diretamente na estrutura de um processo modelo;

função imposta - uma função realizada por um processo função que obtem dados do modelo inspecionando diretamente o vetor de estados do processo modelo;

fusão fixa - uma fusão de duas ou mais sequências de dados de acordo com regras pré-fixadas, e não dependendo de seus valores nos buffers;

fusão forçada - uma fusão de duas ou mais sequências de dados baseada na ordem na qual os buffers das sequências de dados envolvidas tornam-se disponíveis para leitura;

getsv - uma operação executada por um processo que obtém o valor corrente do vetor de estados de outro processo;

inversão - uma transformação na qual um processo sequencial é convertido em um procedimento chamado uma vez a cada buffer de cada sequência de dados com respeito ao qual o processo é invertido;

iteração - um tipo de componente de dados ou de processo que consiste de uma parte que ocorre zero ou mais vezes a cada ocorrência da iteração;

marcador de tempo - um buffer indicando a chegada de um ponto no tempo do mundo real;

modelo - uma abstração realizada especialmente por um conjunto de processos sequenciais;

modelo dinâmico - um modelo envolvendo uma especificação das ações de entidades e a ordem no tempo em que as ações ocorrem;

modelo estático - modelo que não mostra a ordem no tempo;

mundo real - o contexto do qual é extraído o assunto principal do sistema;

nível 0 - o mundo real com o qual o sistema está relacionado;

nível 1 - o modelo direto do mundo real;

passo da estrutura da entidade - o segundo passo no processo de desenvolvimento do JSD, no qual são desenhados os diagramas de estrutura de cada entidade, mostrndo a ordem de ocorrência das ações no tempo;

passo de ações e entidades - o primeiro passo no processo de desenvolvimento do JSD, no qual são listadas as entidades e ações;

passo de função - o quarto passo no processo de desenvolvimento do JSD, no qual são somadas as funções ao sistema;

passo de implementação - o sexto passo no processo de desenvolvimento do JSD, no qual a especificação é transformada para que possa ser devidamente executada;

passo de sincronização - o terceiro passo no processo de desenvolvimento do JSD, no qual é especificada a sincronização adicional necessária para fornecer as saídas corretas do sistema;

passo do modelo inicial - o terceiro passo no processo de desenvolvimento do JSD, no qual são especificados os processos modelos para as entidades do mundo real, e as conexões entre estes processos e as entidades que eles modelam;

ponteiro de texto - uma variável cujo valor indica o ponto, no texto, alcançado pelo processo;

processador - um equipamento usado na implementação, capaz de realizar a execução de um único processo;

processo - execução, do começo ao fim, de um programa ou de uma instância particular de tal execução;

processo função - um processo que é somado ao sistema com o propósito de produzir saídas, em oposição a um processo modelo;

processo modelo - um processo sequencial que é parte de um modelo, em oposição a um processo função;

programadores - as pessoas que realizam o produto para os compradores. Os compradores, usuários, analistas e programadores podem pertencer a mesma organização;

read - uma operação que espera e aceita o próximo buffer de uma sequência de dados de entrada;

seleção - um tipo de componente de processo ou de dado que consiste de duas ou mais partes, das quais ocrre apenas uma a cada ocorrência da seleção;

separação de vetores de estados - uma transformação na qual as variáveis de locais de um processo, incluindo o ponteiro de texto, são separadas da parte executável do texto de processo, armazenadas e acessadas explícitamente como um objeto de dados;

sequência - um tipo de componente de processo ou de dado que consiste de uma ou mais partes, onde cada uma das quais ocorre uma vez a cada ocorrência da sequência;

sequência de dados - um conjunto ordenado de registros ou mensagens através das quais dois processos se comunicam, os registros são lidos por um processo na mesma ordem em que foram escritos por outro processo.

suspenso - esperando por um registro de entrada, e assim é impossível prosseguir até ele tornar-se disponível;

transformação - conversão sistemática de uma parte de um sistema em uma forma de execução mais conveniente e eficiente, sobre um hardware e software disponível;

usuários - as pessoas que operam ou interagem diretamente com o sistema. Os usuários e compradores frequentemente não são as mesmas pessoas;

vetor de estados - as variáveis locais de um processo, incluindo o ponteiro de texto do processo;

write - uma operação que coloca o próximo buffer em uma sequência de dados.

O significado das palavras sublinhadas foi extraído de (IEEE, 1984). As demais palavras pertencem ao glossário de termos do método JSD (JACKSON, 1983).

APÊNDICE B

SINTAXE COMPLETA

Este apêndice mostra a sintaxe de cada um dos comandos aceitos pelo Executor-JSD, onde as palavras reservadas são apresentadas em negrito. Os símbolos colchetes ([]) e chaves ({}) não fazem parte da sintaxe, apenas auxiliam a representação da mesma. Os colchetes delimitam uma parte opcional (não obrigatória) na sintaxe e as chaves indicam uma repetição do trecho delimitado por elas.

. comando de especificação

spec { nome-de-processo : número-de-instâncias lista-de-variáveis } ;

número-de-instâncias
    número

lista-de-variáveis
    (nome-de-variável { ,nome-de-variável })

. comando de sequência

sequence nome-de-sequência : item { ,item } ;

item
    read nome-de-sequência-de-dados [(identificador)]
        to lista-de-variáveis
    read nome-de-sequência-de-dados { & nome-de-sequência-de-dados } **to** lista-de-variáveis
    write nome-de-registro [**to** nome-de-sequência-de-dados [(identifica-dor)]]

getsv nome-de-vetor-de-estados [(identificador)]
to lista-de-variáveis
atribuição
nome-de-estrutura

atribuição
nome-de-variável = expressão-simples

nome-de-estrutura
nome-de-sequência
nome-de-seleção
nome-de-iteração

. comando de seleção

selection nome-de-seleção : lista-de-condições { **alt** lista-de-condições } [**alt**] ;

lista-de-condições
(expressão) item { ,item }

. comando de iteração

iteration nome-de-iteração : lista-de-condições { **quit** lista-de-condições } ;

. comando posit

posit nome-de-posit : lista-de-condições lista-de-quits **admit** lista-de-condições ;

. comando fim

end nome-de-processo ;

. comando de sequência de dados

stream nome-de-sequência-de-dados : conexão ;

conexão

[many] nome-de-processo --> [many] nome-de-processo

. comando de vetor de estados

state nome-de-vetor-de-estados : conexão ;

. comando de sequência de dados externa

external stream nome-de-sequência-de-dados --> nome-de-processo ;

. comando de enquiry

enquiry nome-de-enquiry --> nome-de-processo ;

. comando de marcador de tempo

time nome-de-time --> [many] nome-de-processo ;

. comando de report

report nome-de-processo --> nome-de-report ;

. comando de fusão forçada

rough [**merge**] nome-de-sequência-de-dados { ,nome-de-sequência-de-dados};

expressão

expressão-simples relação expressão-simples

relação

== != < > <= >= || &

expressão-simples

termo { + - || termo }

termo

fator { * / & fator }

fator

nome-de-variável

número

(expressão)

APÊNDICE C

ALGUNS RELATÓRIOS DO DAILY RACKET GERADOS PELO JSD-TOOL

PASSO DE AÇÕES E ENTIDADES:

ENTITIES

leitor
juri

ENTITIES AND ACTIONS

leitor
    assinar,
    competir
juri
    reunir,
    premiar

ACTIONS AND ENTITIES

assinar
    leitor
competir
    leitor
reunir
    juri
premiar
    juri

ATTRIBUTES AND ACTIONS

tipo-de-premio
    premiar
verso
    competir
    premiar
gorgeta
    competir
sequencia
    competir
    premiar
endereço
    assinar
nome
    assinar
data
    assinar
    reunir

RELATÓRIOS DE ANÁLISE

1. POSSIBLE MARSUPIAL ENTITIES:
    No possible marsupial entity.

2. ENTITIES AND NON-SHARED ACTIONS:
    leitor
        assinar,
        competir
    juri
        reunir,
        premiar

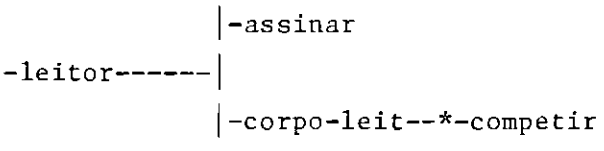
PASSO DE ESTRUTURA DA ENTIDADE:

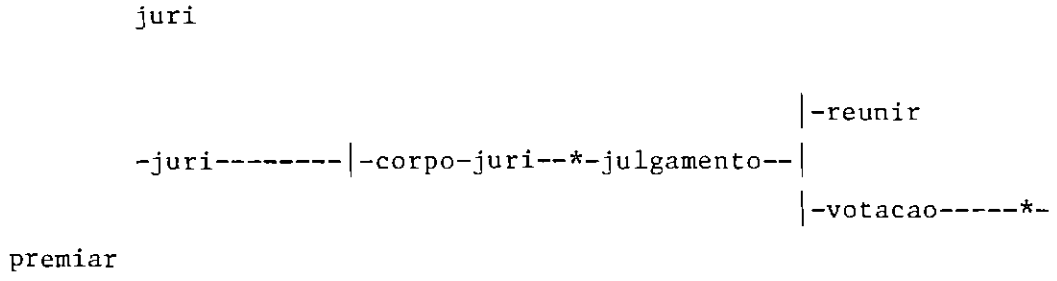
STRUCTURE DEPENDENCES

corpo-leitor
depends on no structure.
leitor
corpo-leitor
corpo-juri
votaçao
julgamento
juri
corpo-juri
julgamento
votaçao
julgamento
votaçao
votaçao
depends on no structure.

ENTITY-STRUCTURE DIAGRAMS

leitor





RELATÓRIOS DE PROBLEMAS

1. CIRCULAR DEFINITIONS

No circular definition.

2. UNDEFINED ENTITIES

No undefined entities.

3. ACTIONS DEFINED AS STRUCTURES

No action defined as structure.

4. NON-ACTION LEAF NODES

No non-action leaf node.

5. INCONSISTENT ENTITY DEFINITIONS

No inconsistent entity definition

PASSO DO MODELO INICIAL:

PROCESSES AND CONNECTIONS

leitor

reads:

L

writes:

no data stream or vector.

juri

reads:

J

writes:

no data stream or vector.

CONNECTIONS

L

external data stream.

sink: leitor

J

external data stream.

sink: juri

CONNECTIVITY ANALYSIS

leitor

reaches:

no internal process.

is reached by:

no internal process.

juri

reaches:

no internal process.

is reached by:

no internal process.

RELATÓRIOS DE PROBLEMAS

1. WELL-FORMEDNESS

CYCLES IN THE INITIAL MODEL

No cycle on initial model.

UNREACHABLE PROCESSES

No unreachable processes.

MULTIPLE EXTERNAL-INTERNAL CONNECTIONS

No multiple connections.

INVALID ROUGH MERGES

No invalid rough merges.

MISSING ITERATION/SELECTION CONDITIONS

```
corpo-leitor
     iter
          corpo-leitor:
               competir,
               read L;
corpo-juri
     iter
          corpo-juri:
               julgamento;
```

CIRCULAR STRUCTURE DEFINITIONS

No circular definition.

2. MODEL VS. ENTITY-ACTION DESCRIPTIONS

NON-ENTITY PROCESSES

Every process is an entity.

MISSSING ENTITIES

No missing entities.

ACTIONS NOT POSSIBLE TO GENERATE

All actions can be generated.

ACTIONS NOT CONSUMABLE BY PROCESSES

All actions can be consumed.

DIRECTLY LINKED PROCESSES NOT SHARING ACTIONS

Not the case.

3. MODEL VS. PROCESS STRUCTURES

MISSING PROCESS STRUCTURES

No missing process structure.

PASSO DE FUNÇÕES: RELATÓRIOS DE PROBLEMAS

1. WELL-FORMEDNESS

USELESS PROCESSES

No useless processes.

UNREACHABLE PROCESSES

No unreachable processes.

INVALID ROUGH MERGES

No invalid rough merges.

MISSING ITERATION/SELECTION CONDITIONS

All conditions are present.

CIRCULAR STRUCTURE DEFINITIONS

No circular definition.

2. SSD VS. PROCESS STRUCTURES

MISSING PROCESS STRUCTURES

No missing process structure.

MISSING read STATEMENTS

All streams are read.

INCORRECT read STATEMENTS

No incorrect read statements.

MISSING write STATEMENTS

All streams are written.

INCORRECT write STATEMENTS

No incorrect write statements.

MISSING getsv STATEMENTS

All state vectors are read.

INCORRECT getsv STATEMENTS

No incorrect getsv statements.