UM SISTEMA PARA EXECUÇÃO DE ESPECIFICAÇÕES JSD

Ana Maria Ambrosio

Flávio Roberto Dias Velasco

Instituto de Pesquisas Espaciais - INPE

Cx.P. 515 São José dos Campos - São Paulo

Sumário

Um dos princípios básicos dos modelos operacionais é a possibilidade de se executar diretamente a especificação de requisitos de software. Neste trabalho, o método JSD ("Jackson System Development") é considerado um modelo operacional e, para tanto, são apresentados uma linguagem e uma ferramenta que possibilitam a execução de especificações JSD.

1- INTRODUÇÃO

No **modelo convencional** de desenvolvimento de software, analistas de sistemas, durante a fase de especificação, formulam e definem um sistema como uma "caixa preta", descrevendo todas as características do seu comportamento externo (o que) e nenhuma característica da estrutura interna (como). Ao contrário, no que se convencionou chamar "**modelo operacional**" (ZAVE 84), a especificação é escrita em uma linguagem formal, executável e independente da implementação (no sentido que suas características possam ser realizadas por uma grande variedade de configurações de recursos). Uma especificação executável é um modelo de um sistema que pode ser executado (sob um executor conveniente) e simular o comportamento do sistema especificado.

Entre os modelos operacionais estão a linguagem de especificação PAISLey (ZAVE 86), Gist (COHEN 83), Descartes (URBAN 85) e, entre outras, a notação gráfica do método "Jackson System Development - JSD" (JACKSON 83), o qual inspirou este trabalho.

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 transformação da rede para rodar sobre um pequeno número de processadores (CAMERON 86).

A proposta deste trabalho é demonstrar a viabilidade dos modelos operacionais no que diz respeito a especificações executáveis. Uma especificação escrita em uma linguagem formal pode ser diretamente executada por um computador, eliminando, com isso, 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 método JSD e implementada uma máquina abstrata (Executor-JSD) capaz de executar tal especificação.

A seção 2 traz uma descrição da contrução do Executor-JSD. Um exemplo de especificação (JACKSON 83) usando a linguagem definida e os resultados da execução da mesma são mostrados na seção 3. Na seção 4 é apresentada algumas conclusões deste trabalho.

2 - EXECUTOR DE ESPECIFICAÇÕES JSD

Nesta seção apresentamos as soluções adotadas para viabilizar a execução de certas características do método JSD no que diz respeito a descrição de uma especificação. O leitor poderá consultar Jackson (83) e Cameron (86) que dão uma boa descrição do método JSD, pois não cabe aqui, a descrição do mesmo.

O primeiro passo para a construção do Executor-JSD foi definir uma sintaxe para a linguagem de especificação que fosse capaz de descrever o Diagrama de Especificação do Sistema (DES) e a estrutura textual dos processos (o que caracteriza uma especificação JSD).

Descrever o DES inclui descrever os dois tipos de conexões permitidas pelo método (sequência de dados e vetor de estados), conexões com o mundo real (entradas e as saídas), multiplicidade de processos e de conexões, fusão forçada e marcador de tempo.

Além de descrever o comportamento do DES e de cada processo, a nova linguagem provê comandos para alocar processos, suas variáveis internas e aumentar o poder expressivo da estrutura textual, sob o ponto de vista da máquina (comandos de atribuição, expressões booleanas, etc).

O Executor-JSD foi criado para executar especificações em microcomputadores com um único processador e sem recursos para multiprogramação, logo, foi necessário elaborar um mecanismo capaz de criar máquinas virtuais para a execução concorrente dos vários processos especificados. Por este lado, ele funciona como um **escalonador**, englobando a função de suspender e continuar a execução dos processos.

O conceito de instâncias de processo aparece aqui como um conjunto de processos do mesmo tipo, também chamados processos múltiplos. Com o objetivo de evitar a repetição de código das várias instâncias de processo, foi mantida apenas uma cópia do texto e tantas cópias do vetor de estados quantas forem as instâncias do processo. O vetor de estados contém o endereço da próxima instrução a ser executada e o conjunto das variáveis do processo.

O mecanismo de conexão por sequência de dados foi implementado através de uma fila (FIFO). As mensagens são inseridas através da execução da operação "**write**" comandadas pelo processo fonte da conexão. O processo destino as obtém através da operação "**read**".

A toda sequência de dados foi associada uma matriz[1], a fim de implementar sua multiplicidade. Cada campo da matriz aponta para uma fila diferente. Para calcular o número de filas que devem ser alocadas para implementação da conexão por sequência de dados com multiplicidade, basta multiplicar o número de instâncias do processo fonte com o número de instâncias do processo destino da conexão. A Figura 2.1 ilustra uma conexão entre dois processos genéricos: P (com três instâncias) e Q (com duas instâncias).

---

1 matriz **nxm** onde **n** é o número de instâncias do processo fonte e **m** é o número de instâncias do processo destino da conexão.

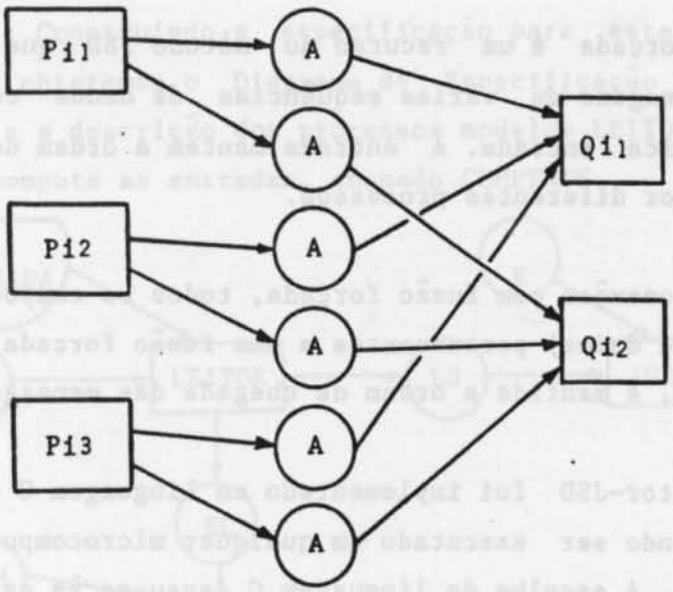

Fig. 2.1 - Implementação de sequência de dados múltiplas.

O acesso às informações de um vetor de estados pelo processo destino de uma conexão é feito através da execução da operação **"getsv"**. A qual fornece uma cópia das variáveis do processo fonte. A multiplicidade das conexões por vetor de estados, sob o ponto de vista da implementação é representada na Figura 2.2, através das instâncias de P e de Q.

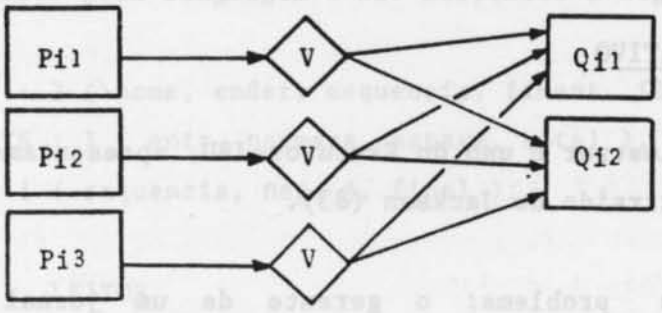

Fig. 2.2 - Implementação de vetores de estados múltiplos.

As entradas e saídas do mundo real são feitas através de instruções de leitura e escrita a uma sequência de dados externa. As mensagens que acompanham tais instruções são emitidas no vídeo, para solicitar uma entrada ou produzir uma saída. Ao encontrar uma instrução de leitura, o Executor emite uma solicitação de dado de entrada e a execução não prossegue até que o usuário responda à solicitação. Para simular a ausência de entrada o usuário deve responder com a mensagem "-".

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. A entrada mantem a ordem de chegada das mensagens, enviadas por diferentes processos.

Nas conexões com fusão forçada, todos os campos de todas as matrizes (sequência de dados) pertencentes a uma fusão forçada apontam para uma única fila. Assim, é mantida a ordem de chegada das mensagens.

O Executor-JSD foi implementado em linguagem C sob o sistema operacional DOS podendo ser executado em qualquer microcomputador com tal sistema operacional. A escolha da linguagem C deveu-se às características econômicas de expressão, controle de fluxo e estruturas de dados.

A técnica de programação seguida foi a de tipos abstratos de dados, pois esta facilita a implementação e a manutenção, melhora a produtividade e a segurança do software, além de permitir a construção de componentes reusáveis (LISKOV 79).

3 - UM EXEMPLO APLICATIVO

Para ilustrar o uso do Executor-JSD, apresentamos nesta seção um exemplo simples extraído de Jackson (83).

Dado o problema: o gerente de um jornal planeja uma competição entre os leitores. Para competir basta ser assinante e enviar uma sequência de entradas cada vez que o jornal publicar detalhes da competição. A entrada consta de três fotografias de modelos enviadas em ordem de elegância.

A competição é julgada, periodicamente, por um júri. As melhores entradas, desde o último julgamento, recebem prêmios. Um relatório semanal, contendo a semana, o número de entradas da semana e o total de entradas, deverá ser fornecido para que o proprietário do jornal possa acompanhar a competição.

Construindo a especificação para este problema, segundo o mètodo JSD, obteremos o Diagrama de Especificação do Sistema mostrado na Figura 3.1 e a descrição dos processos modelos LEITOR e JURI e do processo função que computa as entradas, chamado COMPENTS.

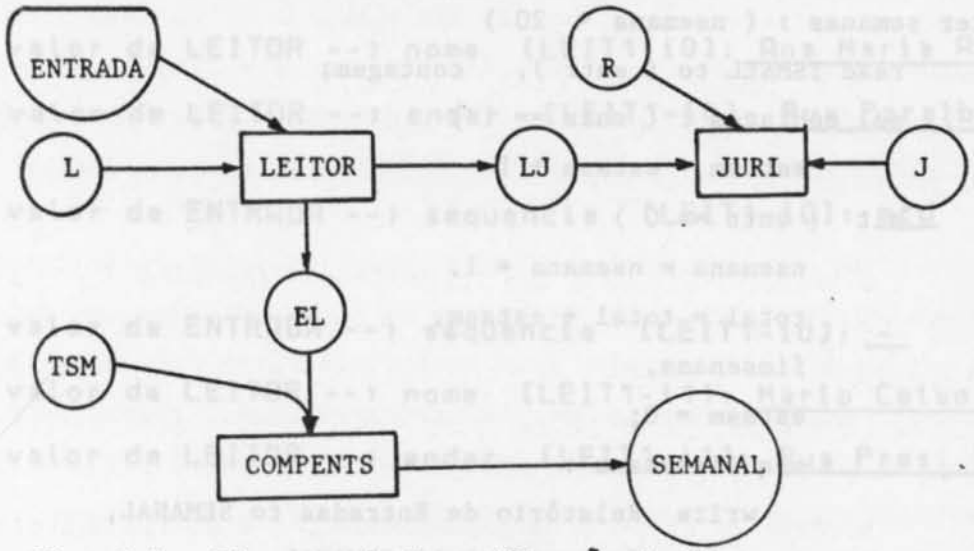

Fig. 3.1 - Diagrama de Especificação do Sistema.

A descrição dos processos è apresentada na linguagem formal definida para poder executar a especificação. O diagrama apresentado acima tambèm è coberto pela linguagem e são mostrados a seguir.

```
spec  LEITOR : 3 ( nome, ender, sequencia, fiment, flag ),
      COMPENTS : 1 ( entr, nsemana, estasm, total ),
      JURI : 1 ( sequencia, neg, j, final );

%   Processo   LEITOR

seq LEITOR :
       read L to ( nome ),  read L to ( ender ),
       read ENTRADA to (sequencia),
       flag = 1,    fiment = fim,
       competir;
       iter competir : ( sequencia != fiment )
              write ( sequencia ) to LJ,
              write ( flag ) to EL,
              read ENTRADA to (sequencia);
end LEITOR;
```

```
%   Processo  COMPENTS

seq COMPENTS :
      total = 0, estasm = 0, nsemanas = 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,
                  fimsenama,
                  estasm = 0;
                  seq fimsemana :
                     write  Relatório de Entradas 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 JURI
% Para simplificar, não descrevemos as funções  referentes ao júri, isto é,
% como  deveria  proceder o julgamento das  sequências de  entrada  e quais
% deveriam ser  as saídas do sistema.

% Definição das conexões

external stream L --> LEITOR;
external stream ENTRADA --> LEITOR;
external stream J --> JURI;
external stream R --> JURI;
stream EL : many LEITOR --> COMPENTS;
stream LJ : many LEITOR --> JURI;
report COMPENTS --> SEMANAL;
time TSM --> COMPENTS;
rough merge TSM, EL;
```

Os primeiros processos executados são as (três) 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.

De o valor de LEITOR --> nome [LEIT1-i0]: Ana Maria Ambrosio

De o valor de LEITOR --> ender [LEIT1-i0]: Rua Paraibuna,55

De o valor de ENTRADA --> sequencia [LEIT1-i0]: acd

De o valor de ENTRADA --> sequencia [LEIT1-i0]: -

De o valor de LEITOR --> nome [LEIT1-i1]: Mario Celso Almeida

De o valor de LEITOR --> ender [LEIT1-i1]: Rua Pres. Dutra, 98

:

O relatório semanal das entradas é emitido após o usuário entrar com o valor 1 no marcador de tempo TSM, indicando o final de uma semana, como mostra o exemplo abaixo.

De o valor de TSM [COMPENTS-i0]: 1

```
Relate SEMANAL -->   Relat de Entradas
Relate SEMANAL -->   .. Semana ..
sm=    1,
Relate SEMANAL -->   Entradas da Semana
estasm=    13,
Relate SEMANAL -->   Total de Entradas
total=    13,
```

4 - CONCLUSÃO

A construção do Executor-JSD contribuiu significativamente para o entendimento dos problemas, vantagens e desvantagens de se executar diretamente especificações, de acordo com o modelo operacional. Em suma, a organização e formalização dos requisitos de um sistema desenvolvido através de um modelo operacional parece ser viável e não absurdamente complexa, porém ainda há uma série de restrições.

No caso do método JSD, as técnicas de especificação são claramente compreensíveis e independentes da implementação. Com relação aos aspectos organizacionais, 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.

O método JSD não apresenta dificuldade ao usuário e o desenvolvimento de um sistema pode tornar-se mais simpático com o auxílio do JSD-tool (VELASCO 86). Desta forma, tornar a especificação um documento escrito em linguagem formal (para que possa ser executada) é uma tarefa trivial.

Como dito anteriormente, o Executor-JSD é uma ferramenta experimental, e portanto, possui uma série de limitações e simplificações. O sistema foi testado apenas com exemplos simples que ilustram o método JSD. 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. Atualmente, planejamos sua aplicação desta ferramenta em sistemas de tempo real desenvolvidos pelo INPE, como por exemplo, software de controle de antena de satélites .

REFERÊNCIAS BIBLIOGRÁFICAS

CAMERON, R.J. An Overview of JSD. IEEE Transaction on Software Engineering, SE-12(2):222-240, Feb. 1986.

COHEN, D. Symbolic Execution of the Gist Specification Language. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE. Karlsruhe, West Germany, Aug. 8-12, 1983. Proccedings, 8. p. 17-20.

JACKSON, M.A. System development. Englewood Cliffs, NJ, Prentice-Hall, 1983.

LISKOV, B. Modular Program Construction Using Abstractions. Cambridge, MA, Laboratory for Computer Science. Massachusetts Institute of Technology. (Computation Structures Group Memo 184) Sep. 1979, p. 2-43.

URBAN, S.D.; URBAN, J.E.; DOMINICK W.D. Using an executable specification language for an information system. IEEE transactions on Software Engineering, SE-11(7): 598-605, July 1985.

VELASCO, F.R.D. JSD-Tool reference manual. Tyngsboro, MA. Wang Institute Graduate Studies, 1986. (TR-86-09).

ZAVE, P. The Operational versus the Conventional Approach to Software Development. Comunications of ACM, 27(2):104-118, Feb. 1984.

ZAVE, P.; SCHELL, W. Salient features of an executable specification language and its environment. IEEE Transaction on Software Engineering, SE-12(2):312-325, Feb. 1986.