UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UMA ABORDAGEM PARA A REPRESENTAÇÃO, SIMULAÇÃO E IMPLEMENTAÇÃO DE SISTEMAS BASEADA NA REDE DE PETRI A OBJETOS

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA PARA OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA

EVANDRO CANTÚ

FLORIANÓPOLIS, MARÇO DE 1990.

UMA ABORDAGEM PARA A REPRESENTAÇÃO, SIMULAÇÃO E IMPLEMENTAÇÃO DE SISTEMAS BASEADA NA REDE DE PETRI A OBJETOS

EVANDRO CANTÚ

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA ELÉTRICA

ESPECIALIDADE **ENGENHARIA ELÉTRICA**, ÁREA DE CONCENTRAÇÃO **SISTEMAS DE CONTROLE**, E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

Prof. JEAN MARIE FARINES, Dr. Ing.
Orientador

Prof. JOSÉ CARLOS MOREIRA BERMUDEZ, Ph. D.
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

BANCA EXAMINADORA

Prof. JEAN MARIE FARINES, Dr. Ing.
Orientador

Prof. HERVE ERIC GARNOUSSET, Dr.
Co-orientador

Prof. JOSÉ EDUARDO R. CURY, Dr. D'Etat

Profª. STEFANIA STHUBIENER, Dra.

À Lu e Anna Lu

AGRADECIMENTOS

Ao Prof. Jean Marie Farines pela orientação e dedicação ao longo do desenvolver deste trabalho, ao Prof. Hervé Garnousset pelo apoio e contribuição, ao Celso Kaestner pelo auxílio, aos membros da Banca Examinadora e aos demais professores e colegas do LCMI pela amizade e convívio neste período. Também agradeço a CAPES pelo auxílio financeiro em forma de bolsa.

SUMÁRIO

RESUMO

A presente dissertação visa propor, para o formalismo Rede de Petri a Objetos (RPO), metodologias e ferramentas a serem utilizadas no desenvolvimento de aplicações para Automação Industrial e Sistemas Distribuídos.

São apresentados os principais aspectos sintáticos e semânticos de uma linguagem de descrição formal de sistemas baseada na Rede de Petri a Objetos. Tal linguagem permite descrever o comportamento dinâmico do sistema e representar os dados na forma de objetos, além de possuir características de estruturação e modularidade.

Apresenta-se também um interpretador para a execução das especificações em RPO, baseado na analogia do algoritmo de evolução da marcação da Rede de Petri com o Mecanismo de Inferência de um Sistema de Produção (SP) de primeira ordem. Este interpretador pode ser utilizado para a simulação, prototipagem e implementação direta das especificações, através de um mecanismo de inferência baseado em um algoritmo de filtragem eficiente.

ABSTRACT

This dissertation suggests, for the Petri Net with Objects (PNO) model, methodologies and tools to be used on the development of Industrial Automation and Distributed Systems Applications.

The main sintactic and semantic aspects of a system formal description language based on PNO are presented. Such language supports the dinamic behavior description and data representation as objects, and have structure and modularity features.

We show the similarity between the net marking evolution algorithm and first order Production System (PS) Inference Mechanism. From this analogy, a PNO interpreter which can be used on simulation, prototyping and specification implementation is built, based on an efficient incremental unification algorithm.

CAPÍTULO 1

INTRODUÇÃO

Este trabalho visa desenvolver um conjunto de metodologias e ferramentas para auxiliar no desenvolvimento de Sistemas de Automação Industrial e de Sistemas Informáticos Distribuídos.

Os Sistemas de Automação Industrial visam integrar de forma automatizada os diversos níveis de decisão dentro do ambiente industrial, envolvendo atividades que vão desde o controle local de uma máquina, até o planejamento [Valette, 1986]. O projeto e implementação destes sistemas é uma tarefa complexa; cada nível hierárquico no qual o sistema é decomposto apresenta problemas distintos a serem resolvidos, porém estes problemas tem uma interdependência, existindo portanto a necessidade de se integrar cada um desses níveis de forma a haver uma perfeita coerência entre a produção e o planejamento da empresa.

Seguindo as tendências atuais da informática, os Sistemas Informáticos Distribuídos permitem a decentralização dos centros de decisão, oferecendo aos utilizadores as vantagens de compartilhamento de recursos, melhoria do desempenho graças ao paralelismo e de melhoria da disponibilidade global do sistema [Courtiat, 1987]. Esses sistemas são utilizados em muitos campos de aplicação, dentre os quais em aplicações ligadas à Automação Industrial.

Estas duas áreas de aplicação apresentam alguns pontos em comum, em particular no que diz respeito ao alto grau de paralelismo e comunicação entre seus componentes. Como veremos no desenvolver deste trabalho, esta característica permitirá pensar na solução dos problemas encontrados em ambas as aplicações, através de uma mesma técnica.

O processo de desenvolvimento de sistemas e em especial para as aplicações citadas, visa sempre garantir uma implementação correta e eficiente dos requisitos fixados

pelo usuário. Para solucionarmos este problema, é indispensável que o desenvolvimento desses sistemas baseie-se numa Técnica de Descrição Formal (TDF), de forma a descrever de forma completa, clara e sem ambigüidades as especificações. A utilização de TDF permite oferecer uma base para uma metodologia de desenvolvimento e para a construção de ferramentas automatizadas para auxílio em cada uma das etapas desta metodologia.

A Fig. 1 apresenta as várias etapas da metodologia de desenvolvimento de sistemas adotada neste trabalho. Ela consiste primeiramente em descrever as especificações informais do sistema usando um modelo formal. A partir do modelo formal obtido, pode-se analisar e validar estas descrições, de modo a poder detectar erros de concepção. Informações sobre o comportamento do sistema e a possibilidade de uma avaliação de desempenho, também podem ser obtidas através de técnicas de análise e simulação. Em seguida, a partir das especificações validadas, e através de um procedimento muitas vezes automatizado, será possível a implementação. Finalmente, realiza-se o teste de conformi-

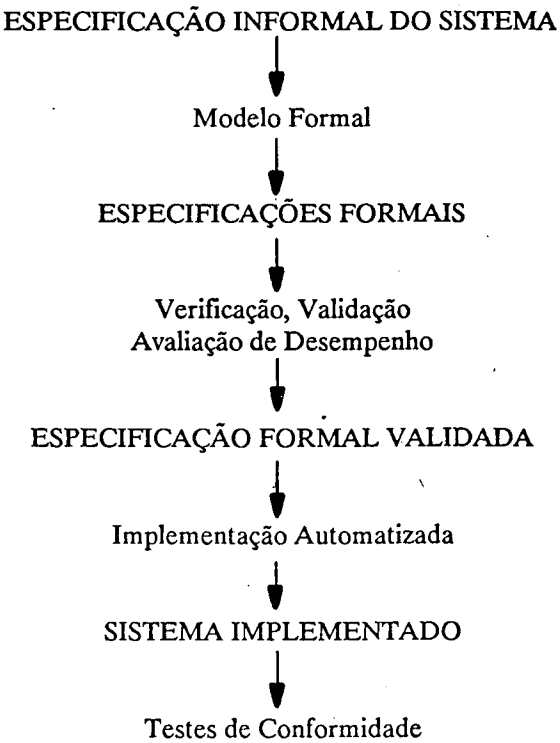


Fig. 1. Etapas no Desenvolvimento de um sistema.

dade da implementação resultante com a especificação inicial. Ao final de cada uma destas etapas pode haver revisões, quando erros, incoerências ou mau desempenho forem detectados.

Dentre os diversos modelos formais nos quais as Técnicas de Descrição Formal (TDF) se baseiam, a Rede de Petri (RdP) foi o modelo escolhido, sendo que neste trabalho, utilizar-se-á uma das suas extensões, a Rede de Petri a Objetos (RPO) [Sibertin-Blanc, 1985].

O problema da representação de sistemas é introduzido no segundo capítulo. Após ter levantado as características principais das TDF, justifica-se a escolha da Rede de Petri a Objetos (RPO) como modelo de base para as aplicações de interesse neste trabalho.

No terceiro capítulo é proposta uma linguagem de representação de sistemas baseada no formalismo Rede de Petri a Objetos (LRPO), através da qual introduzimos os conceitos de estruturação e modularidade. A sintaxe e a semântica desta linguagem são apresentadas informalmente neste capítulo.

No quarto capítulo é apresentado um simulador (SRPO) para sistemas descritos pela linguagem baseada na Rede de Petri a Objetos, vista anteriormente. O simulador desenvolvido baseia-se na execução de um Sistema de Regras equivalente ao modelo de Rede de Petri. O mecanismo de execução é similar à Máquina de Inferência de um Sistema de Produção (SP) de primeira ordem. Neste trabalho, será apresentado um algoritmo para melhorar a eficiência deste mecanismo. A abordagem proposta oferece suporte para a simulação, prototipagem e implementação direta das especificações do sistema. Finalmente, serão também discutidas algumas das características que o algoritmo oferece para facilitar a análise das especificações e para possibilitar a implementação em Ambientes Distribuídos.

No último capítulo apresentar-se-ão conclusões e perpectivas do trabalho.

CAPÍTULO 2

MODELAGEM DE SISTEMAS COMPLEXOS

2.1. Introdução

Neste capítulo será discutido o problema da representação de Sistemas Complexos, com particular interesse nas áreas de aplicação de Sistemas de Automação Industrial e de Sistemas Informáticos Distribuídos, procurando identificar um modelo formal que possa satisfazer os dois tipos de aplicações.

Após ter levantado as principais características das Técnicas de Descrição Formal (TDF) e dos formalismos nos quais elas estão baseadas, justifica-se a escolha da Rede de Petri (RdP) como modelo de base para este trabalho. Serão apresentadas as características da Rede de Petri e discutida algumas extensões deste modelo, em especial a Rede de Petri a Objetos (RPO), utilizada neste trabalho, devido ao grande potencial deste modelo para aplicações nas áreas já citadas.

2.2. As Técnicas de Descrição Formal (TDF)

O primeiro passo da metodologia de desenvolvimento adotada neste trabalho consiste em descrever o comportamento dos sistemas de maneira completa, consistente, concisa, sem ambigüidades e precisa. As Técnicas de Descrição Formal (TDF) tem cumprido este papel, sendo reconhecido que as mesmas devem apresentar requisitos de expressividade, abstração, grau de formalização matemática e estruturação [ISO/DIS, 1987, 1988].

A expressividade de uma TDF é entendida como a capacidade de representar todas as características e conceitos encontrados nas aplicações de interesse (Automação

Industrial e Sistemas Distribuídos). Destaca-se em particular todas as noções associadas a representação de sistemas com paralelismo, como [Bougé, 1988]:

- a sequencialidade (dependência causal);
- a concorrência (independência causal);
- o conflito (uma causa pode ter vários efeitos diferentes);
- a sincronização (várias causas independentes devem ser produzidas antes que o efeito seja percebido);
- e a comunicação (transferência de informação).

As estruturas de dados, restrições de tempo e de desempenho devem também poder ser expressas [Bruijning, 1987].

A característica de abstração de uma TDF permite que em cada etapa do desenvolvimento, o usuário possa se abstrair de detalhes irrelevantes para esta etapa e garantir uma independência com relação a implementação.

O formalismo matemático suportado pelas TDF deve oferecer plenas condições para a análise das especificações, para a prototipagem e implementação, para determinar a conformidade da implementação em relação a especificação inicial e, para a construção das ferramentas correspondentes.

Outrossim, as TDF devem apresentar características de estruturação que permitem a construção de sistemas através da composição de sub-sistemas, facilitando igualmente a legibilidade das especificações.

Existe hoje um grande número de abordagens, ou modelos de base, para as TDF, baseadas em formalismos diversos, dos quais podemos destacar [Courtiat, 1987]:

- as *máquinas de estado finita* (MEF), onde o sistema é em geral representado na forma de uma composição de máquinas de estado finitas descrevendo o comportamento de cada parte do sistema, com a interconexão de cada parte podendo ser através de mecanismos de filas FIFO ("First In, First Out") ou diretamente por fusão de transições; as técnicas de análise são baseadas na análise do grafo de acessibilidade resultante;
- as *redes de Petri*; esta abordagem será estudada em detalhe na seção seguinte;

- as *gramáticas formais*, onde as frases geradas pela gramática correspondem a seqüências válidas de eventos caracterizando o sistema; as técnicas de validação são fundamentadas sobre as propriedades formais das gramáticas;
- a *álgebra de processos* (p.ex. abordagem algébrica CCS), onde o sistema é descrito por uma equação de comportamento; as técnicas de validação fazem referência a equivalência observacional entre modelos;
- os *tipos abstratos de dados*, onde por regras de escrita e esquemas de dedução é possível, de maneira semi-automática verificar as propriedades da especificação;
- a *lógica temporal*, que permite por exemplo verificar sequências válidas de eventos a partir de asserções da lógica temporal;
- e *formalismos híbridos*, baseados nos formalismos anteriores.

Em particular cabe destacar os esforços de padronização de Técnicas de Descrição Formal efetuados pela ISO (linguagens Estelle e Lotos) e pelo CCITT (linguagem SDL), na área de Sistemas Distribuídos e Protocolos de Comunicação. Lotos é fundamentada na álgebra de processos CCS e nos tipos abstratos de dados. Estelle e SDL são fundamentadas na máquina de estado finita, comunicando-se por fila FIFO; Estelle faz uso de estruturas da linguagem PASCAL para a representação dos dados e SDL representa os dados por tipos abstratos algébricos e possui uma forma gráfica normalizada [Courtiat, 1987; Diaz, 1989].

2.3. A Escolha da Rede de Petri como Modelo

Dentre os diversos formalismos existentes para TDF, a Rede de Petri (RdP) foi o modelo escolhido, motivado pelo seu poder de expressão (em particular na representação do paralelismo), abstração, facilidade de uso e principalmente pela capacidade de construção de ferramentas automatizadas para a verificação, validação e avaliação de desempenho, fornecida pelo seu formalismo.

2.3.1. O modelo de rede de Petri

A Rede de Petri (RdP) [Peterson, 1981] é um tipo particular de grafo orientado, constituído de dois tipos de nós: as transições que correspondem aos eventos que caracterizam as mudanças de estado do sistema e os lugares que correspondem as condições que devem ser verificadas para os eventos acontecerem. A presença (ou não) de fichas num lugar indica a verificação (ou não) da condição associada a este lugar. A distribuição das fichas nos lugares do modelo indica o estado, sendo chamada marcação.

A Fig. 2 ilustra a regra de disparo das transições da RdP.

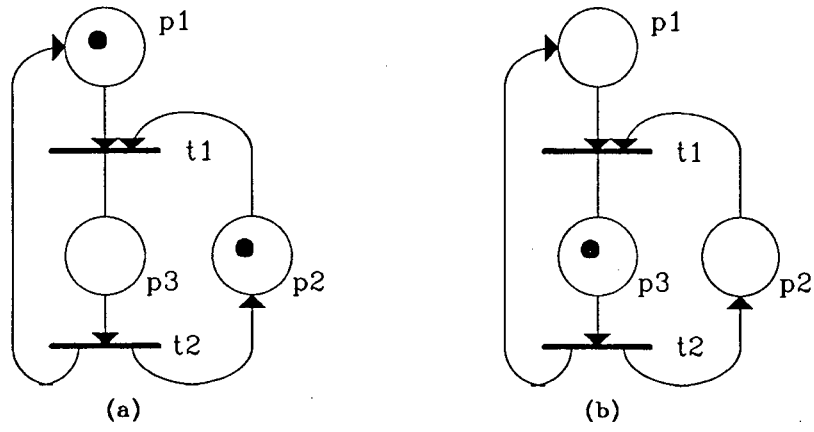


Fig. 2. Regra de disparo da RdP: a) antes do disparo; b) depois do disparo.

2.3.2. Justificativas para a Escolha da Rede de Petri

A partir dos requisitos que as TDF devem apresentar, apontamos a seguir as qualidades do modelo RdP para ser utilizado como formalismo de base para uma descrição formal e também suas limitações, indicando para estas últimas as possibilidades de contorná-las.

Primeiramente, a expressividade do modelo de RdP permite descrever precisamente as relações de causalidade encontradas nas aplicações consideradas, como a sequencialidade (Fig. 3a); o conflito (Fig. 3b); a concorrência (Fig. 3c) e a sincronização

(Fig. 3d). As restrições de tempo e possibilidade de avaliação de desempenho também podem ser obtidas a partir do uso de extensões como as Redes de Petri Temporizadas [Merlin, 1976], ou outras. As limitações de expressividade da RdP vem no sentido da representação dos dados, sendo necessário a introdução de outras estruturas que permitam uma representação mais adequada dos dados do sistema.

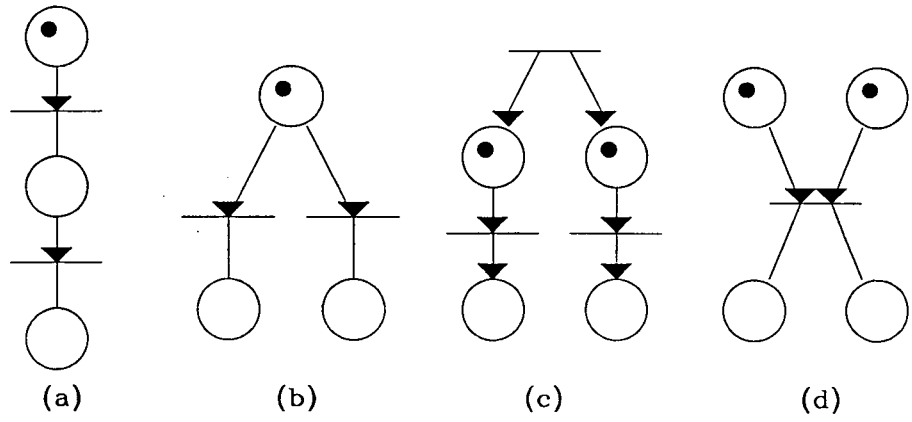


Fig. 3. Relações de causalidade modeladas por RdP: a) sequencialidade; b) conflito; c) concorrência; d) sincronização.

A RdP apresenta um bom nível de abstração em comparação com outros modelos, o que garante uma independência em relação a implementação.

O formalismo matemático no qual a RdP baseia-se permite a análise das especificações, sendo possível a análise das propriedades gerais do modelo (correção), das propriedades específicas (comportamento) e a avaliação do desempenho, conforme veremos a seguir.

Um dos métodos de análise da RdP está baseado na geração do grafo de estados acessíveis da rede, que quando é finito, permite fornecer as respostas às propriedades gerais (limitação, vivacidade, reinicialização) ou específicas (exclusão mútua, acessibilidade,...). Existem várias técnicas de análise do grafo de acessibilidade, baseadas no conceito de equivalência, na noção de projeção e sobre a lógica temporal [Diaz, 1989].

A RdP permite também realizar a análise estrutural, através da determinação dos invariantes de lugar e de transição da rede, sendo possível verificar algumas propriedades

específicas, como a conservação, a acessibilidade e as sequências cíclicas de disparo de transições. Esta forma de análise é interessante pois não necessita gerar o grafo de estados, cuja limitação se deve a explosão do espaço de estados no caso de redes complexas. A simulação é também uma opção de interesse para uma verificação parcial das propriedades do modelo de RdP.

Para a avaliação do desempenho, a associação de densidades de probabilidades ao intervalo de disparo das transições da RdP (Rede de Petri Temporizada Extendida) permite avaliar o desempenho temporal do sistema descrito, através de medidas como tempo médio entre eventos ou a probabilidade de ocorrência de eventos [Roux, 1987].

Para estas técnicas de análise da RdP, existem muitas ferramentas automatizadas. Ao nosso alcançe, no Laboratório de Controle e Microinformática (LCMI) foi desenvolvido um Ambiente para Concepção de Sistemas baseado na RdP (ARP) [Farines, 1989b] que apresenta um conjunto de ferramentas de análise, simulação e avaliação de desempenho. A Fig. 4 apresenta as ferramentas oferecidas pelo ARP.

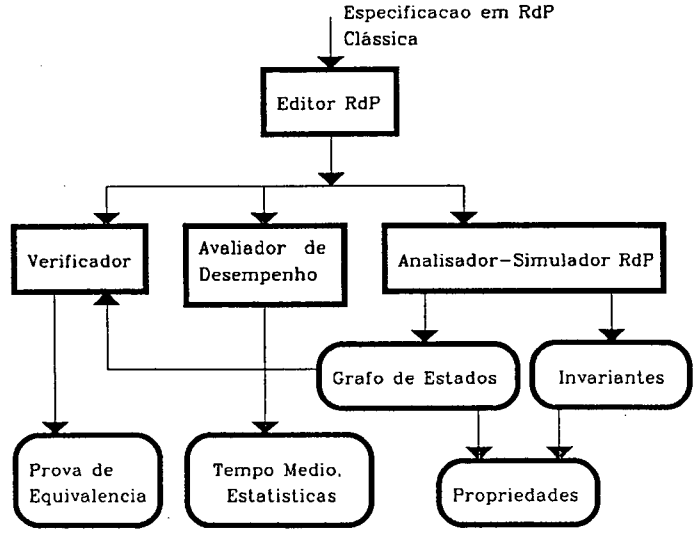


Fig. 4. Conjunto de Ferramentas do Ambiente ARP.

Outrossim, o modelo RdP não oferece possibilidades de estruturação das especificações, sendo necessário neste caso apresentar soluções externas ao formalismo.

Todas as características apresentadas mostram as potencialidades da RdP para ser utilizada como modelo de base de uma TDF. Os pontos negativos da RdP contudo são a falta de expressividade no que se refere a descrição dos dados e a dificuldade de estruturação. Partindo destas limitações, existe a necessidade de se extender o formalismo RdP introduzindo uma forma de representação dos dados, que será apresentada na seção 2.4, e propor extensões que permitem estruturar a especificação, sendo este um dos objetivos deste trabalho conforme será visto no capítulo 3.

2.3.3. Uso Atual da Rede de Petri

Em aplicações de Automação Industrial, a RdP tem boa aceitação, sendo utilizada com maior ênfase nos níveis mais baixos da automatização. Dos cinco níveis hierárquicos comumente encontrados nos Sistemas de Automação Industrial (o controle local; a coordenação de sub-sistemas; a coordenação global e monitoração em tempo-real; o ordenamento e o planejamento), a RdP vem sendo utilizada nos três primeiros. No controle local, o modelo RdP, na sua forma clássica ou através da representação GRAFCET, é utilizado na programação do controle de equipamentos [Valette, 1986]. No nível de coordenação de sub-sistemas, a RdP e suas extensões permitem modelar de forma eficiente sistemas de transporte, Células Flexíveis de Usinagem etc. Neste caso, os lugares da RdP representam estoques de peças, grupo de máquinas, pontos de recepção de mensagens etc; às fichas podem ser associadas estruturas de dados relativas a peças, ferramentas ou máquinas, ou mensagens trocadas entre tarefas, o que torna necessário o uso de extensões que permitam esta associação [Alanche, 1984; Valette, 1986; Bako, 1989].

Nas aplicações em Sistemas Distribuídos, a RdP têm sido utilizada para especificar e validar protocolos de comunicação. Os mecanismos de comunicação (síncrona ou assíncrona) e as restrições de tempo, comumente empregados na modelização de protocolos, podem ser representados a partir deste modelo ou das suas extensões [Diaz, 1982, 1989; Courtiat, 1987].

2.4. Rede de Petri a Objetos

2.4.1. Extensões à Rede de Petri Clássica

A modelagem completa de um Sistema de Automação Industrial ou de um Protocolo de Comunicação necessita de uma maior expressividade no que se refere a descrição dos dados, do que àquela suprida pelas RdP clássicas que representam apenas a estrutura de controle do sistema. Estas limitações da RdP clássica foram compensadas a partir de extensões, que permitem a individualização das fichas fazendo com que as mesmas possam transmitir uma informação. A Rede de Petri Colorida, a Rede Predicado/Transição (PrT) [Genrich, 1987], a Rede de Petri Numérica [Billington, 1988] e a Rede de Petri a Objetos (RPO) [Sibertin-Blanc, 1985] são as principais propostas de extensões.

i) Rede de Petri Colorida

No modelo de Rede de Petri Colorida as fichas podem ser distingüidas através de uma cor, que indica a sua identidade. Assim, a cada lugar, bem como a cada transição, é associado explicitamente um conjunto de cores. Os arcos são etiquetados por funções das cores da transição nas cores do lugar.

Para exemplificar, a Fig. 5a [Diaz, 1982] mostra o exemplo do Almoço dos Filósofos modelado por uma RdP clássica. No caso apresentado, um filósofo pode comer quando dois palitos quaisquer estiverem livres. Esta solução contudo não resolve o problema de que um filósofo somente pode comer quando estiverem livres os palitos imediatamente a sua direita e esquerda. Para uma solução correta usando RdP clássica será necessário representar cada filósofo e cada palito por um lugar, conforme apresentado em [Peterson, 1981]. Usando a Rede de Petri Colorida, conforme apresentado na Fig. 5b, obtemos uma solução correta e mais compacta; para a transição t1 disparar, deve existir um filósofo f_i no lugar p1 (filósofo pensando) e dois palitos $\{p_i$ e $p_{i+1}\}$ no lugar p2 (palitos livres).

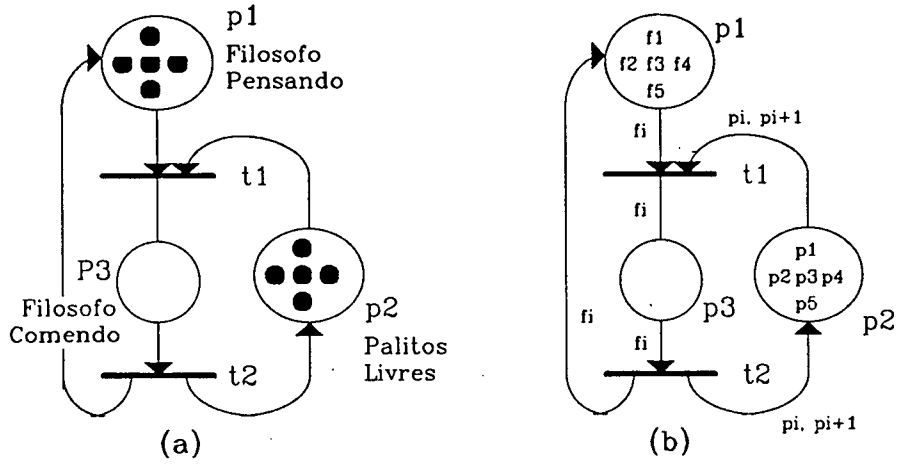


Fig. 5. Problema do Almoço dos Filósofos: a) modelado por RdP clássica; b) modelado por Rede de Petri Colorida.

ii) Rede Predicado/Transição (PrT)

A Rede Predicado/Transição (PrT) [Genrich, 1987] é um dos modelos de RdP de maior utilização atualmente, sendo empregada em várias áreas - em Protocolos de Comunicação destaca-se os trabalhos de Azema [1989] e Lloret [1987], utilizados nesta dissertação. Ao nível conceitual, a rede PrT é fundamentada no formalismo matemático da lógica de predicados de primeira ordem. Neste modelo, o conceito de ficha colorida é extendido, sendo permitido aos lugares conter fichas parametradas por n-uplas de constantes e variáveis. Estas n-uplas guardam a identidade de cada um de seus elementos e também representam claramente as relações dinâmicas entre os mesmos. Os arcos são etiquetados por somas formais de variáveis definindo a configuração de fichas que será consumida ou produzida em um lugar. Sobre as transições podem ser escritas fórmulas lógicas utilizando as variáveis dos arcos de entrada das transições. Assim, para a transição disparar, ela deve estar sensibilizada por uma substituição consistente das variáveis pelas n-uplas que constituem a marcação e o predicado associado deve ser verificado.

Um exemplo de disparo para a rede PrT é dado na Fig. 6.

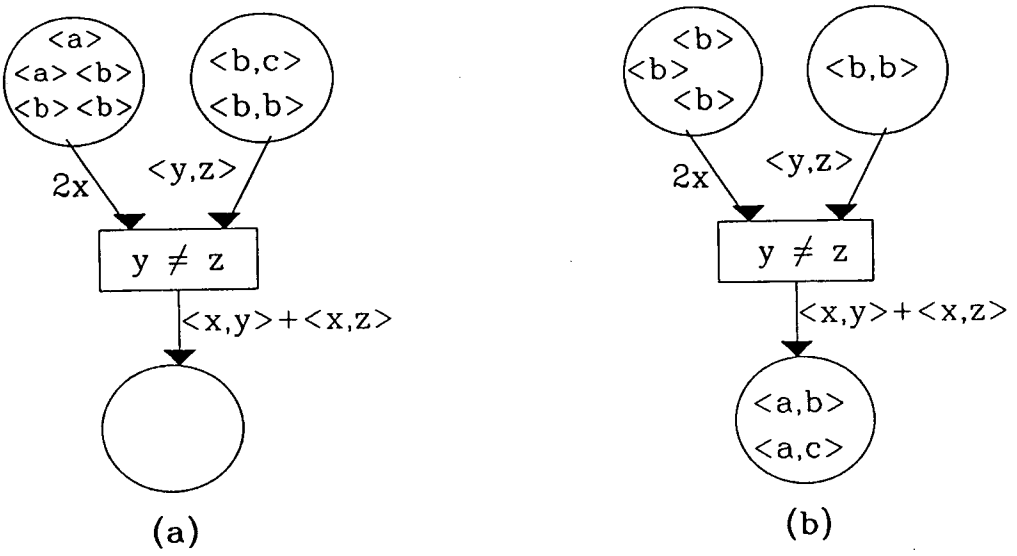


Fig. 6. Disparo na Rede Predicado/Transição: a) antes do disparo; b) depois do disparo.

iii) Rede de Petri Numérica

A Rede de Petri Numérica é uma extensão às redes PrT, utilizada sobretudo para modelagem de Protocolos de Comunicação [Billington, 1988]. Neste modelo é possível associar um conjunto de variáveis do tipo dado (inteiro, booleano, string etc) a rede. Estas variáveis permitem modelar objetos como dados de controle, contadores etc, e podem ser manipuladas através de condições e ações associadas a cada transição. O teste a 0 (inexistência de ficha em um certo lugar) também pode ser representado de maneira explícita. Na Rede de Petri Numérica, os arcos apresentam três formas de inscrições: 1) condições que devem ser satisfeitas pela coleção de n-uplas (fichas) associada aos lugares de entrada; 2) n-uplas que serão removidas dos lugares de entrada; e 3) n-uplas que serão criadas nos lugares de saída quando a transição disparar. Estas inscrições diferenciam a condição de sensibilização, da movimentação de fichas durante o disparo

iv) Rede de Petri a Objetos (RPO)

A Rede de Petri a Objetos (RPO) [Sibertin-Blanc, 1984, 1985, 1988] é um modelo semelhante ao modelo da rede PrT, mas que utiliza a noção de objetos para a representação dos dados. Conceitualmente ela é derivada da teoria de base de dados relacionais e das linguagens orientadas a objetos, o que a torna um formalismo mais acessível ao usuário menos habituado à lógica de predicados de primeira ordem. Em particular, a RPO é adequada para a modelagem de aplicações ligadas à Automação Industrial. Nestas aplicações o uso da noção de objetos permite uma melhor estruturação dos dados do sistema e facilita o mapeamento direto dos objetos físicos manipulados [Vallete, 1988].

No modelo de RPO, a marcação é dada por fichas individualizadas, representadas por Objetos, que modelam os elementos manipulados pelo sistema. Cada Objeto é identificado por um nome e é considerado como uma instância da Classe de Objetos a qual pertence. A fim de identificar os objetos sem ambigüidades, cada um deles terá um atributo identificador e cada objeto terá um valor diferente para este atributo.

Uma Classe de Objetos é uma estrutura com um nome de classe e uma lista de dimensão variável de propriedades (ou atributos), cada uma com um domínio de valores.

A cada lugar da RPO, é associado um conjunto de n-uplas de classes de objetos, e somente instâncias de objetos destas n-uplas podem ser depositadas ou removidas deste lugar.

Para poder manipular os objetos, é declarado um conjunto de variáveis, explicitando para cada variável a classe de objetos que ela poderá capturar.

As transições da RPO permitem selecionar os objetos nos lugares de entrada através das variáveis sobre os arcos. O disparo destas transições dependem da existência desses objetos e de condições a serem verificadas sobre os objetos capturados. Se as condições forem satisfeitas, então a transição pode ser disparada, o que resulta em ações

que modificam a semântica desses objetos e em uma nova distribuição destes nos lugares de saída da transição.

A Fig. 7 ilustra a forma gráfica da RPO, mostrando a parte de condição e ação de uma transição.

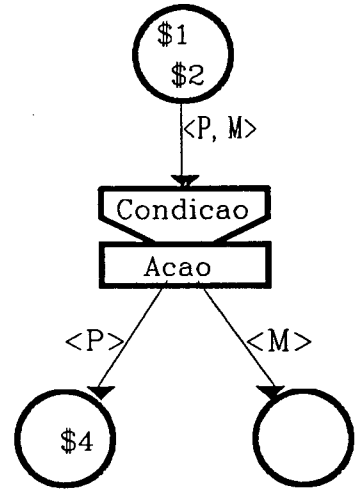


Fig. 7. Forma gráfica da RPO.

Uma definição formal da RPO é dada no QUADRO 1.

Dentre os modelos citados, optou-se pela Rede de Petri a Objetos, e justificamos sua escolha nos seguintes pontos. Primeiro, por apresentar uma melhor estruturação dos dados do sistema através da noção de objetos. Esta característica permite um mapeamento direto dos objetos físicos no caso de aplicações em Automação Industrial. Também nas aplicações em Protocolos e Sistemas Distribuídos existe um interesse crescente na utilização da noção de objetos. Segundo, pela facilidade que a noção de objetos oferece para a construção de ferramentas automatizadas. E finalmente, pelo fato de ser um modelo próximo da rede PrT, que é um modelo muito utilizado.

QUADRO 1. Definição Formal da RPO [Siberin-Blanc, 1988]

Seja **O** um conjunto de objetos, notaremos por **O*** o conjunto das n-uplas de objetos, e por $P(O^*)$ o conjunto das partes de **O***.

Definição 1: Uma RPO é definida por:

1) Um conjunto finito **CO** de Classes de Objetos.
2) Um conjunto finito **T** de transições.
3) Um conjunto finito **P** de lugares, e uma função cl_P:**P**-->$P(CO^*)$, que especifica os conjuntos de n-uplas de **CO** que cada lugar pode conter;
4) Um conjunto **V** de variáveis e uma função cl_V:**V**-->**CO**, que especifica a Classe de Objeto que cada variável pode capturar;
5) Uma função de incidência de entrada: **Pre: P x T --> P(V*)**
   e uma função de incidência de saída: **Pos: P x T --> P(V*)**
   que respeitam a dimensão e a classe dos lugares;
   se **v ∈ Pre(t,p)** então **v** (resp. **p**) é uma variável (resp. lugar) de entrada de **t**;
   se **v ∈ Pos(t,p)** então **v** (resp. **p**) é uma variável (resp. lugar) de saída de **t**.
6) A cada transição pode-se associar um conjunto de predicados, chamados condições, cujas variáveis sejam as variáveis de entrada de **t**.
7) A cada transição pode-se associar um conjunto de ações, da forma:
   v.atr <-- Expr onde:
   v é uma variável de saída de **t**;
   atr é um atributo desta variável;
   Expr é uma expressão onde suas variáveis são as variáveis de entrada de **t**.
8) Um conjunto de registros globais, que são partilhados por todas as transições, podendo figurar nas condições e nas ações das transições.

Definição 2: Seja **O** um conjunto de identificadores para os objetos. Um estado da RPO é chamado marcação e é definido por:

1) Uma função distributiva **m: P --> $P(O^*)$**
   que indica para cada lugar os conjuntos de n-uplas de objetos que ele contém;
2) Um valor para cada atributo dos objetos que figuram nos lugares.

Definição 3: Seja **M** uma marcação e **t** uma transição de uma RPO, **t** é dita sensibilizada pela marcação **M**, se existe objetos que são disponíveis nos seus lugares de entrada e que satisfaçam as condições associadas a **t**. Ou mais formalmente, se existe uma substituição **S: V --> O**, tal que:

1) para todo lugar **p**, **S(Pre(t,p))** $\subseteq$ **M(p)**,
2) se $Pc(v_1,...,v_n)$ é a pré-condição de **t**, se a variável v_i captura o objeto o_i (i=1..n), então a proposição $Pc(o_1,...,o_n)$ é verdadeira.

O disparo de uma transição sensibilizada depois da marcação **M**, leva a nova marcação **M'**. O estado **M'** é atingido pelos seguinte passos:

1) Retirar dos lugares de entrada de **t** as n-uplas de objetos capturados pelas variáveis;
2) Criar um novo objeto para cada variável de saída de **t**, que não seja ao mesmo tempo uma variável de entrada, de tal forma que cada variável capture um objeto;
3) Executar em paralelo cada uma das ações de **t**;
4) Depositar nos lugares de saída de **t** as n-uplas de objetos capturados pelas variáveis de saída.

2.4.2. Exemplo de Modelagem de uma Célula Flexível de Usinagem por RPO

Para ilustrar a modelagem por RPO, vamos considerar o exemplo de aplicação em Automação Industrial, apresentado na Fig. 8. O sistema representa uma Célula Flexível de Usinagem, formada por um par de Máquinas de Comando Numérico (M1 e M2), que processam peças transportadas por uma mesa rotativa composta de quatro bandejas ("pallets") nas quais as peças são depositadas. As peças são retiradas de um estoque e carregadas na mesa na posição 1, e descarregadas após o término da usinagem na posição 4 e depositadas no mesmo estoque. As peças são usinadas pela máquina M1 (ou máquina M2) quando presentes na bandeja da posição 2 (respectivamente posição 3). A cada peça é associado um conjunto de tarefas ("job"), definido por uma sequência de usinagem sobre as máquinas M1 e M2, como por exemplo a seqüência (M1 M2 M1).

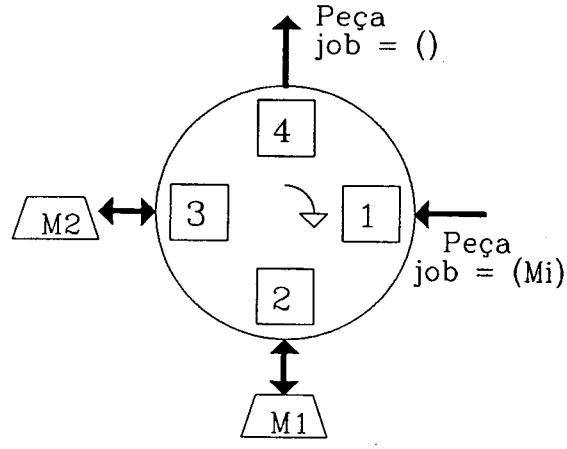


Fig. 8. Célula Flexível de Usinagem.

O modelo de RPO equivalente é mostrado na Fig. 9, onde o lugar p1 representa o estoque das peças, p2 representa a mesa rotativa, p3 representa as máquinas livres e p4 representa as máquinas usinando. As transições ENTRADA e SAÍDA representam as operações de carga e descarga das peças da mesa. A rotação da mesa é representada pela transição ROTAÇÃO, e o início e fim de usinagem pelas transições INICIO e FIM. Na figura, para cada transição é apresentada sua parte de condições e ações.

No exemplo, identificamos três Classes de Objetos (Peça, Pallet e Máquina), escolhidas pelo simples mapeamento dos objetos físicos do sistema. Associado ao gráfico da RPO, é apresentado na Fig. 9 a declaração das classes de objetos com seus respectivos atributos (precedidos pelo símbolo "^"), dos lugares e das variáveis. A marcação inicial é representada pela distribuição dos objetos nos lugares da rede, identificados pelo símbolo $n, mas os valores para os atributos dos objetos devem ser dados separadamente.

2.5. Conclusão

Neste capítulo foi tratado o problema da representação dos Sistemas Complexos. Após a descrição das características e das várias abordagens utilizadas pelas Técnicas de Descrição Formal, foi escolhido o modelo de Rede de Petri como formalismo de base para este trabalho.

Para fazer face as limitações de expressividade da Rede de Petri no que se refere a representação dos dados foi apresentado o modelo Rede de Petri a Objetos, escolhido para este trabalho devido a sua capacidade de representar uma estrutura de dados mais complexa e pelo interesse da representação de objetos, tanto a nível de modelagem para aplicações em Automação Industrial e Sistemas Distribuídos, como para o desenvolvimento das ferramentas associadas.

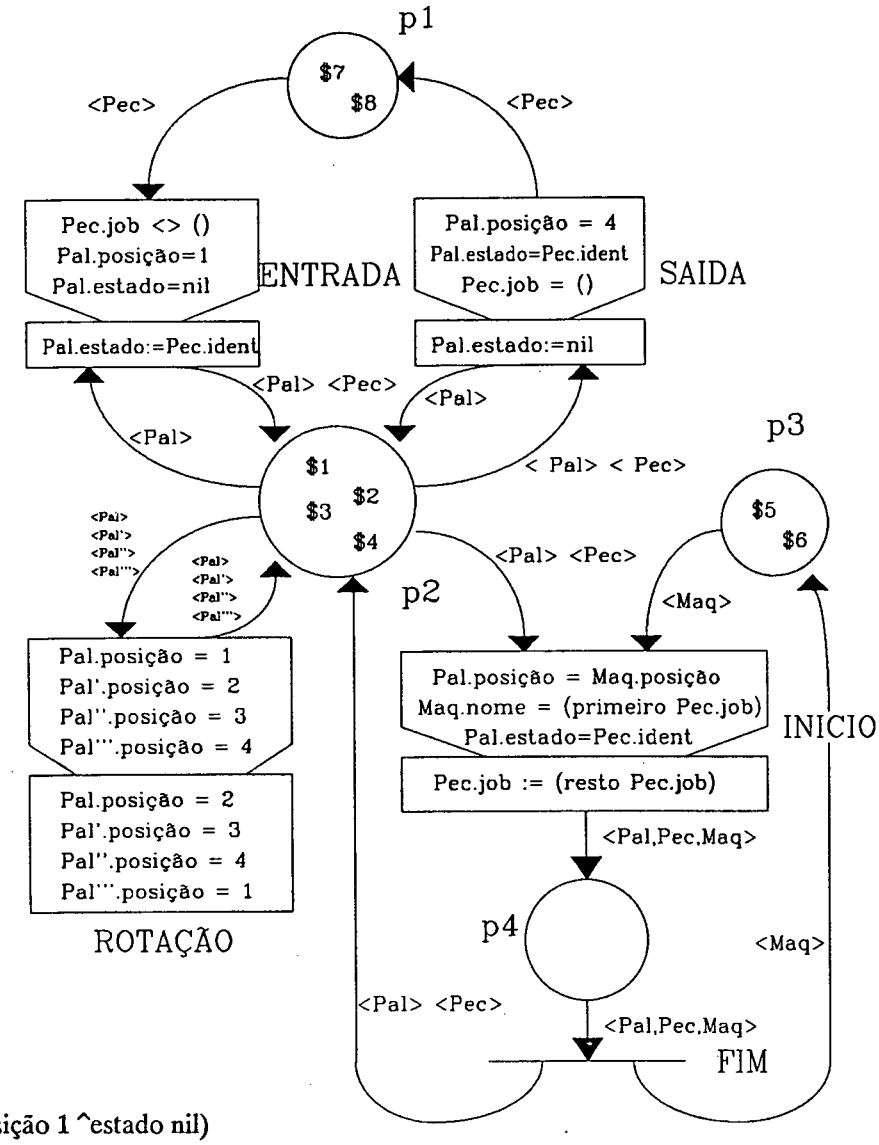
As limitações do modelo de RPO no que se refere a estruturação serão tratadas no capítulo seguinte, onde está sendo proposta uma linguagem baseada na Rede de Petri a Objetos (LRPO), através de uma extensão deste modelo.

CO = {Peça, Pallet, Máquina}
T = {ENTRADA, SAÍDA, ROTAÇÃO, INÍCIO, FIM}
P = {p1, p2, p3, p4}
V = {Pec, Maq, Pal, Pal', Pal", Pal"'}

Atributos:
Peça -> ^ident ^job
Pallet -> ^posição ^estado
Máquina -> ^nome ^posição

cl_V:
Pec : Peça
Maq : Máquina
Pal, Pal', Pal", Pal"'

cl_P:
p1 : <Peça>
p2 : <Pallet>, <Peça>
p3 : <Máquina>
p4 : <Pallet, Peça, Máquina>



Marcação:
$1 : (Pallet ^posição 1 ^estado nil)
$2 : (Pallet ^posição 2 ^estado nil)
$3 : (Pallet ^posição 3 ^estado nil)
$3 : (Pallet ^posição 4 ^estado nil)
$5 : (Máquina ^nome M1 ^posição 2)
$6 : (Máquina ^nome M2 ^posição 3)
$7 : (Peça ^ident P1 ^job (M1 M2 M1))
$8 : (Peça ^ident P2 ^job (M2 M1))

Fig. 9. Visão da RPO com sua Marcação Inicial para a Célula Flexível.

CAPÍTULO 3

PROPOSTA DE UMA LINGUAGEM PARA ESPECIFICAÇÃO DE SISTEMAS BASEADA NA REDE DE PETRI A OBJETOS (LRPO)

3.1. Introdução

Neste capítulo são apresentadas as características principais da Linguagem de Especificação de Sistemas proposta neste trabalho (LRPO), que tem como formalismo de base a Rede de Petri a Objetos (RPO).

A necessidade de atender aos requisitos de estruturação e modularidade, desejados numa Técnica de Descrição Formal (FDT), motivou a proposição feita neste trabalho de uma linguagem que além do formalismo Rede de Petri a Objetos, descrito no capítulo anterior, inclui estas características.

Após ter discutido a introdução dos conceitos de estruturação e modularidade nas linguagens de especificação, será descrita a sintaxe e a semântica da linguagem LRPO. Dois exemplos de modelagem são apresentados para facilitar o entendimento da linguagem. No final do capítulo algumas perspectivas de evolução e modificação da linguagem são discutidas.

3.2. Estruturação e Modularidade

Quando os sistemas são muito complexos ou quando suas componentes são distribuídas fisicamente, uma das formas de realizar o projeto consiste em construir o sistema global a partir da composição de sub-sistemas, que podem ser refinados sucessivamente e interligados entre si por um mecanismo de comunicação. Este paradigma de decomposição modular é adotado em outras TDF, como na linguagem Estelle

[Budkowski, 1987] e na linguagem modular baseada na rede Predicado/Transição proposta em [Lloret, 1988].

A linguagem proposta LRPO segue também o princípio da decomposição modular descrito anteriormente e se apresenta na forma de uma estrutura de módulos organizados hierarquicamente conforme apresentado na fig. 10a [Budkowski, 1987].

Cada módulo é ligado com outros módulos através da sua interface, formada de portos de entrada/saída (chamados também de pontos de interação em outras linguagens). Dois tipos de ligação são possíveis entre eles, conforme apresentado na Fig. 10b:

1) Ligação de dois módulos irmãos, para representar a comunicação (linhas cheias);

2) Ligação de um módulo com seu módulo pai, para representar a hierarquia (linhas tracejadas).

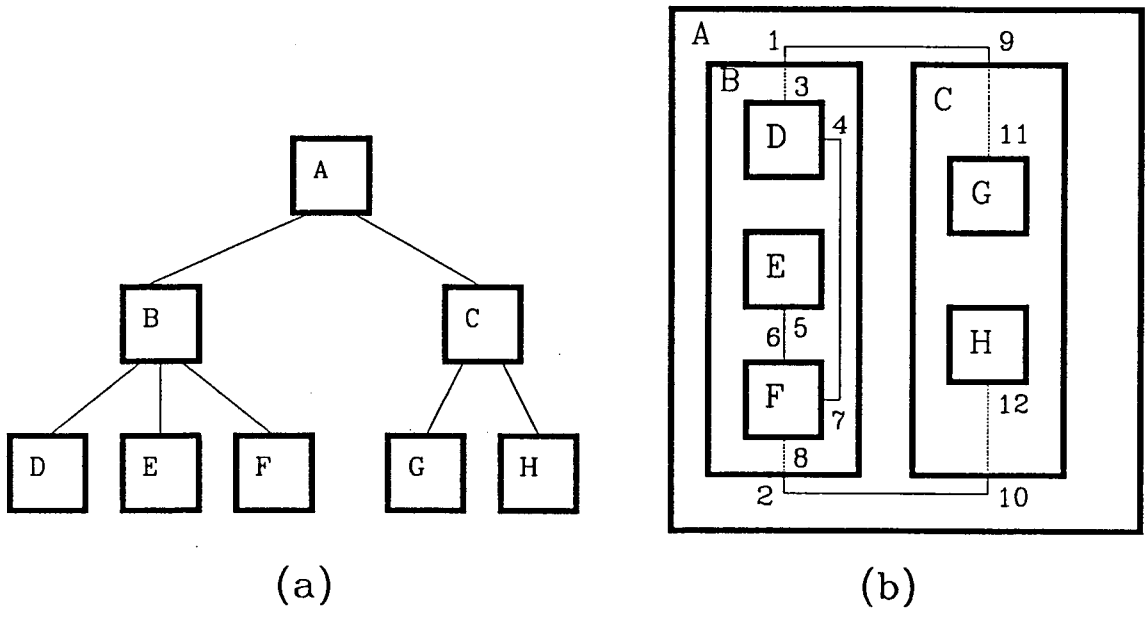


Fig. 10. Estruturação Modular: a) hierarquia de módulos; b) ligações possíveis entre os módulos.

O mecanismo de comunicação adotado para a troca de mensagens entre os módulos é a comunicação síncrona, do tipo "rendez-vous". A escolha do "rendez-vous" se justifica a partir dos seguintes pontos:

- este mecanismo permite a composição entre os módulos de forma similar ao utilizado no formalismo CCS [Lloret, 1988];

- permite representar outros tipos de interações entre módulos, inclusive a comunicação por fila, simulando o funcionamento da mesma por um módulo especial [Courtiat, 1988];
- o "rendez-vous" impõe restrições entre as entidades comunicantes, restringindo de fato a combinatória do produto de seus conjuntos de estados o que facilita a verificação [Lloret, 1988].

O comportamento de cada módulo é descrito por uma Rede de Petri a Objetos, desta forma, cada comunicação síncrona pode ser interpretada de maneira simples pela fusão de transições. A representação da fusão de transições será introduzida na linguagem através de cláusulas de sincronização na guarda de cada transição a ser fusionada.

A linguagem de especificação proposta LRPO, permite então descrever uma especificação na forma de uma estrutura hierárquica de módulos, cujo corpo é uma Rede de Petri a Objetos, com a composição entre os módulos sendo realizada por um mecanismo de sincronização que permite fusionar as transições.

Concluindo, a linguagem LRPO inclui os formalismos Rede de Petri a Objetos para descrição do comportamento da especificação e das estruturas de dados manipuladas, permitindo desta forma utilizar as ferramentas de análise/simulação permitidas pela RdP e introduzir outras extensões sobre o modelo inicial (como: temporização, densidades de probabilidades etc). Do ponto de vista da composição entre módulos, a linguagem LRPO apresenta similaridade com os formalismos CCS e Lotos, o que facilita a verificação. Do ponto de vista da inclusão na linguagem de características de estruturação e modularidade, leva a linguagem LRPO a uma similaridade sintática e semântica com a linguagem Estelle [Budkowski, 1987]. A seguir a sintaxe e semântica da linguagem LRPO serão apresentadas.

3.3. Apresentação da linguagem - Sintaxe e Semântica

A descrição sintática da linguagem é apresentada na forma BNF (Backus Naur Form) simplificada. Nesta BNF os símbolos não terminais serão encerrados por "<" e por ">" e os elementos em negrito caracterizam caracteres e símbolos definidos na linguagem. Também são utilizados os meta-símbolos "::=" (é definido por), "*" (repetição zero ou mais vezes), "+" (repetição uma ou mais vezes), "|" (ou), "{" e "}" (que cercam elementos opcionais).

3.3.1. Tipos de Dados Utilizados na Linguagem

i) Objetos

Os dados básicos manipulados na linguagem são Objetos físicos ou conceituais. Cada objeto é uma instância de uma Classe de Objetos (**CLASS**) caracterizada através lista de dimensão variável de propriedades (ou atributos); cada propriedade é definida por um nome e por um valor. Por exemplo, o objeto:

(Peca ^nome p1 ^tamanho 1/2 ^quantidade 10)

é uma instância da Classe de Objetos Peça.

Como será visto adiante, dentro de cada módulo da estrutura, devem ser declaradas as Classes de Objetos a serem utilizadas, caracterizando desta forma a estrutura dos dados manipulados.

Uma Classe de Objetos é defininda pelo nome da classe e por um conjunto de propriedades (ou atributos) que caracteriza os objetos desta classe.

<declaração_classe_objetos> ::=

(**CLASS** (*<nome_classe_objeto>* ^*<nome_atributo>* *)+)

onde:

<nome_classe_objeto> é o nome da Classe de Objeto;

<nome_atributo> é o nome de um atributo da Classe de Objeto.

Obs: Na frente do nome do atributo coloca-se o sinal "^", para diferenciar corretemente os atributos da sua classe e dos seus valores.

ii) Registros

Além de objetos, podemos definir um certo número de registros (**REGISTER**) que contenham um valor. Estes registros são considerados objetos globais para o módulo no qual são declarados e para todos os módulos internos a este. Estes registros são utilizados para representar dados de controle, declarar constantes, ou outros indicadores do processo (relógios etc). Os valores dos registros podem ser modificados pelas transições da RPO associada ao módulo.

<declaração_registros> ::=

(**REGISTER** *<registro>* +)

onde:

<registro> ::= @*<nome_registro>* *<valor>*

<nome_registro> é o nome do registro;

<valor> é um valor numérico ou simbólico qualquer, ou outro registro já definido.

Obs: A declaração de um registro deve ser sempre acompanhada por uma atribuição de um valor inicial para o mesmo. O nome do registro é pré-fixado por "@", visando sua fácil identificação dentro da linguagem.

iii) Variáveis

Dentro da linguagem variáveis (**VAR**) podem ser utilizadas, para fazer referência a objetos que poderão ser instanciados dentro das transições ou pelas primitivas de comunicação dos módulos.

Todas as variáveis a serem utilizadas dentro de um módulo deverão ser declaradas nele. A declaração de variáveis vai especificar um nome de variável e a classe de objetos

que esta variável poderá capturar. Cada variável poderá capturar objetos de uma única classe.

<declaração_variáveis> ::=

(**VAR** *<variável>*+)

onde:

<variável> ::= *<nome_variável>* *<nome_classe_objeto>*

<nome_variável> é o nome da variável.

3.3.2. Estruturação e Modularidade

Na linguagem LRPO, uma especificação pode estar constituída por um único módulo, ou por um conjunto de módulos aninhados de forma hierárquica. Várias instâncias de um mesmo módulo podem ocorrer numa mesma especificação.

i) Módulo Raiz

A estruturação de uma especificação é iniciada com a declaração de seu módulo raiz (**STRUCTURE**) que representa a estrutura do sistema especificado. Este módulo não possui interface externa e assume-se que existe somente uma instância do mesmo, sendo constituído por uma parte de declaração de dados e de um corpo.

O corpo do módulo raiz consiste na declaração de eventuais módulos filhos e em uma configuração inicial para sua estrutura modular através da definição das suas instâncias de módulos filhos ativas e as ligações entre estas. No caso da ausência de estrutura modular, este corpo pode conter somente uma parte de transições que descreve o comportamento do sistema global na forma de uma Rede de Petri a Objetos (RPO), conforme será visto a seguir.

<declaração_módulo_raiz> ::=

(STRUCTURE *<nome_estrutura>*

<declaração_dados>

<corpo_da_estrutura>)

onde:

<declaração_dados> ::=

{*<declaração_classe_objetos>*}

{*<declaração_registros>*}

{*<declaração_variáveis>*}

<corpo_da_estrutura> ::=

<declaração_modulo> +

<instanciação> *

<ligação_connect> *

|

<rede_de_petri_objeto> .

ii) Módulo

Por sua vez, cada módulo (**MODULE**) contém, além da declaração de dados internos e de seu corpo, como no módulo raiz, uma interface que caracteriza a visibilidade externa do mesmo e através da qual é realizado o acesso ao módulo.

<declaração_módulos> ::=

(MODULE *<nome_módulo>*

<declaração_dados>

<interface_do_módulo>

<corpo_do_módulo>) .

A interface do módulo consiste na declaração de um conjunto de portos de entrada/saída (**PORTS**) com os respectivos tipos de interações permitidos em cada um deles em termos de emissão (**OUT**) ou de recepção (**IN**).

Cada declaração do tipo de interação é identificada pelo nome desta (identificador da interação) e eventualmente por um parâmetro, representado por uma variável, que vai indicar o tipo da mensagem (i.e. definir a classe de objetos) que poderá ser trocada entre as entidades conectadas, através deste porto.

<interface_módulo> ::=

(**PORTS** *<declaração_porto>*+)

onde:

<declaração_porto> ::=

(*<nome_porto>* **IN** (*<tipo_interação>**)

OUT (*<tipo_interação>**))

<tipo_interação> ::=

<nome_interação> {(*<nome_variável>*+)}

A declaração do corpo do módulo é similar à do corpo do módulo raiz em termos de composição: declaração de módulos filhos; configuração inicial com a definição das instâncias filho ativas, das ligações entre elas e, das ligações entre estas instâncias filho e o módulo pai; ou uma parte descrevendo o comportamento do mesmo através de uma RPO. No caso de uma estrutura modular estática, cada módulo terminal da hierarquia (módulo folha) conterá uma parte de transições - a possibilidade de se associar uma RPO a um módulo não terminal não é utilizada, pois a mesma pode ser sempre definida em um módulo filho [Lloret, 1988].

<corpo_módulo> ::=

<declaração_modulo>+

*<instanciação>**

*<ligação_connect>**

*<ligação_attach>**

|

<rede_de_petri_objeto>

A definição das instâncias de módulos ativas (**INSTANCE**) é feita com a atribuição de um módulo a cada uma das instâncias criadas.

<*instanciação*> ::=

(**INSTANCE** <*nome_instância*> **WITH** <*nome_módulo*>)

As ligações entre as instâncias de módulos vão permitir a abertura de canais de comunicação entre elas, sendo possível somente dois tipos de ligação, citados anteriormente:

1) Ligação para comunicação (**CONNECT**) entre duas instâncias de módulos de mesmo nível hierárquico e filhos do mesmo pai (módulos irmãos) (na Fig. 10b os pares (5,6), (4,7), (1,9) e (2,10) ligados por linhas cheias). É necessário que os portos dos dois módulos a serem conectados sejam de sentido oposto, pois as interações emitidas por um módulo deverão ser recebidas pelo outro;

<*ligação_connect*> ::=

(**CONNECT** <*nome_instância*>.<*nome_porto*>

TO <*nome_instância*>.<*nome_porto*>)

2) Ligação para hierarquização (**ATTACH**) entre um módulo e seu módulo pai estabelecendo uma relação ascendente/descendente (na Fig. 10b os pares (1,3), (2,8), (9,11) e (10,12) ligados por linhas tracejadas). É necessário que os portos dos dois módulos a serem ligados sejam de mesmo sentido, pois a operação consiste somente numa renomeação de um porto do pai para o seu filho;

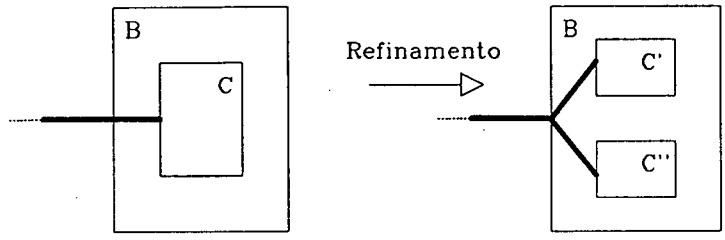
<*ligação_attach*> ::=

(**ATTACH** <*nome_porto_pai*>

TO <*nome_instância*>.<*nome_porto*>) .

A existência de um **CONNECT** e zero ou mais **ATTACH** estabelece um canal de comunicação entre módulos, sendo que somente aqueles portos que são pontos terminais de conexão podem trocar mensagens através deste (na Fig. 10b os portos (3,11), (5,6), (4,7) e (8,12)).

No caso da construção ATTACH, é possível estabelecer ligação entre um módulo pai e várias instâncias filho, desde que os tipos de interações possíveis em termos de envio e recepção no módulo pai, correspondam exatamente a união de todas as interações em termos de envio e recepção nas instâncias filho. Esta extensão permite que a introdução de refinamentos em um módulo não afete as declarações de portos nos níveis superiores. A Fig. 11 ilustra este caso.




```
(MODULE modB
   ...
   (PORTS ipB IN  (Rec(X))
              OUT (Env(Y) Rec(Z)) )
   ...
   (MODULE modC
      PORTS ipC IN  (Rec(X))
                OUT (Env(Y) Rec(Z)) )
   ...
   )
   (INSTANCE instC WITH modC)
   (ATTACH ipB TO instC.ipC)
)
```

(a)

```
(MODULE modB
   ...
   (PORTS ipB IN  (Rec(X))
              OUT (Env(Y) Rec(Z)))
   ...
   (MODULE modC
      (PORTS ipC1 IN  (Rec(X)))
             ipC2 OUT (Env(Y) Rec(Z)))
   ...
   )
   (INSTANCE instC' WITH modC)
   (INSTANCE instC'' WITH modC)
   (ATTACH ipB TO instC'.ipC1,instC''.ipC2)
)
```

(b)

Fig. 11. Exemplo da ligação ATTACH: a) módulo original; b) refinamento.

3.3.3. Rede de Petri a Objetos Associada ao Módulo

Cada módulo da estrutura tem seu comportamento representado por uma Rede de Petri a Objetos [Sibertin-Blanc, 1985] que faz parte do corpo do módulo. Para descrever completamente a RPO associada ao corpo do módulo, a linguagem LRPO contém as seguintes declarações:

- classes de objetos e registros que a rede poderá manipular.

- variáveis, declarando as variáveis que serão utilizadas para instanciar os objetos nas transições e indicando a classe de objetos que cada uma pode capturar;

- lugares, indicando para cada lugar da rede as n-uplas de classes de objetos que o mesmo pode conter;

- conjunto de transições que descrevem o comportamento do módulo;

- marcação inicial para a rede, indicando os objetos que cada lugar contém e os valores dos mesmos.

i) Classe de Objetos, Registros e Variáveis

A declaração de classe de objetos, de registros e de variáveis da rede de Petri a objetos associada ao módulo segue a semântica e a sintaxe vistas anteriormente, e devem ser declarados na área de declaração de dados do módulo.

ii) Classes de Lugares

Cada lugar da RPO poderá conter objetos de uma única classe, ou uma n-upla de objetos que conservam entre si uma relação temporária, ou um conjunto destas n-uplas.

A declaração dos lugares **(PLACE)**, permite indicar o que cada lugar pode conter e tem a seguinte sintaxe:

<declaração_lugar> ::=

(PLACE *<lugar>* +)

onde:

<lugar> ::=

<nome_lugar> (*<n-upla_objeto>* +)

<n-upla_objeto> ::= < *<nome_classe_objeto>* + >

Obs: Nesta declaração, cada n-upla de classes de objetos é encerrada por "<" e ">" (Estes sinais aparecemem negrito para não haver confusão com aqueles da BNF).

iii) Conjunto de Transiçoes

Cada transição da rede de Petri a objetos associada ao módulo pode ser vista como uma regra do tipo SE pré-condição ENTÃO pós-condição, e possui a seguinte sintaxe:

<transição> ::=

(**TR** *<nome_transição>* **PRE** *<pré-condição>* **POS** *<pós-condição>*)

onde a parte de pré-condição é formada por uma função de incidência de entrada que permite selecionar os objetos nos lugares de entrada, e um conjunto de condições que são predicados que devem ser verificados pelos valores dos atributos dos objetos selecionados; e a parte de pós-condição é formada por um conjunto de ações que permitem modificar os valores dos atributos desses objetos e por uma função de incidência de saída que distribui os objetos nos lugares de saída.

A linguagem LRPO foi influenciada em sua sintaxe das transições, pela linguagem de representação de regras em Sistemas de Regras de Produção SP1 [Kaestner, 1989], desenvolvida no LCMI. Desta manteve-se a noção predicativa, acrescendo de outras características inerentes a Rede de Petri a Objeto.

- Pré-Condição da transição

A parte pré-condição da transição inicia por uma seleção de objetos nos lugares de entrada das transições. Esta seleção é feita com o auxílio das variáveis declaradas anteriormente. O objeto selecionado terá seus atributos testados pelas condições da transição.

A sintaxe da parte de pré-condição da transição é a seguinte:

<pré-condição> ::= *<seletor_objetos>* +

<condição> *

onde:

<seletor_objetos> ::=

<nome_lugar> (*<n_upla_variável>* +)

onde:

<n_upla_variável> ::= < *<nome_variável>* + >

<condição> ::=

^*<nome_atributo>* (*<nome_variável>*) *<predicado>* *<termo>*

onde:

<predicado> pode ser um predicado definido na linguagem como =, < >, >, > =, <, < =, Member, Or, And, Not ou outro predicado definido pelo usuário;

<termo> pode ser uma constante, um atributo de um objeto ou uma função sobre outros termos, e serve como parâmetro para avaliação dos predicados.

- Pós-Condição da transição

A parte de pós-condição da transição é composta de um conjunto de ações a serem avaliadas em sequência quando a transição for disparada e de uma nova distribuição dos objetos nos lugares de saída.

Os objetos a serem depositados nos lugares de saída são de dois tipos:

- objeto selecionado na pré-condição, quando aparece na pós-condição a variável utilizada para seleção deste objeto;
- objeto novo, quando atribuirmos a um lugar de saída uma nova variável.

O aparecimento de uma variável apenas na pré-condição da transição indica o consumo pela transição do objeto instanciado.

As ações poderão atribuir valores aos atributos de um objeto que está sendo criado, como também modificar os atributos dos objetos intânciados na parte de pré-condição da transição manipulando os mesmos através das variáveis.

A sintaxe da pós-condição da transição é a seguinte:

<pós-condição> ::= *<distribuição_objetos>* +

<ação> *

onde:

<distribuição_objetos> ::=

<nome_lugar> (< *<n_upla_variável>* + >)

<ação> :=

^*<nome_atributo>* (*<nome_variável>*) = *<termo>*

Aos atributos de um objeto recém criado podem ser atribuídos valores constantes, ou uma função de qualquer atributo que a transição tenha acesso.

Os registros globais, pré-fixados por "@" podem ser utilizados tanto na parte pré-condição como na parte de pós-condição das transições.

- Transições agrupadas

A linguagem permite agrupar transições em conflito efetivo, isto é, transições que possuam os mesmos lugares de entrada e condições sobre os mesmos objetos, sendo estas condições exclusivas umas em relação as outras. Neste caso, a linguagem contém um seletor de objetos comum às transições agrupadas; as condições sobre os objetos são agrupadas (**COND.. OR.. OR..**) de forma similar a um *case* do PASCAL ou a um *cond* do LISP.

A sintaxe das transições agrupadas se apresenta da seguinte forma:

(**TR** *<nome>*

PRE *<seletor_objetos>* +

COND *<condicão>*

POS *<pós-condição>*

OR *<condição>*

POS *<pós-condição>*)

iv) Marcação inicial

A marcação inicial (**INIT**) consiste em uma atribuição de objetos aos lugares da RPO com seus recpectivos valores.

<marcação_inicial> ::=

(**INIT** *<iniciação>* +)

onde:

<iniciação> ::= *<nome_lugar>* (*<objeto>* +)

<objeto> ::=

(*<nome_classe_objeto>* *<atributo>* *)

onde:

<atributo> ::= ^*<nome_atributo>* *<valor>*

v) Funções definidas pelo usuário

A linguagem permite também que o usuário construa qualquer função em linguagem LISP, para utilizá-la posteriormente dentro da transição. O protótipo de simulador apresentado no capítulo 4 suporta esta possibilidade.

3.3.4. Mecanismo de Comunicação

Como visto anteriormente, o mecanismo escolhido para a comunicação entre módulos é do tipo síncrono ("rendez-vous"), que é representado no caso das Redes de Petri pela fusão de transições.

A linguagem LRPO tem cláusulas especiais de emissão e recepção que servem como guarda nas transições, permitindo ao serem fusionadas as transições, a comunicação síncrona entre os módulos. A sintaxe desta guarda é próxima daquela definida no formalismo CSS; são identificados o porto, o sentido da comunicação ("!" para envio e "?" para recepção) e o tipo de interação que será trocado entre os módulos.

A sintaxe da guarda no caso do envio:

SINC *<nome_porto>* **!** *<tipo_interação>*

no caso da recepção:

SINC *<nome_porto>* **?** *<tipo_interação>*

Duas transições com guarda para a comunicação, poderão ser fusionadas se:
- toda a parte pré-condição das duas transições forem satisfeitas;
- existir um canal de comunicação, formado pela conexão dos portos de cada transição, sendo verificado a identidade dos tipos de interação dos portos.

O disparo sincronizado das transições sincronizadas é realizado através troca de parâmetros e da execução paralela das pós-condições das duas transições.

- Extensões sintáticas ao "rendez-vous"

A linguagem LRPO admite a possibilidade sintática de associar a cada transição, ou uma cláusula unitária, ou duas cláusulas, uma em recepção e outra em envio. Desta forma, a linguagem aumenta seu poder de representação. Ao nível semântico, algumas considerações são necessárias.

Suponha que tenhamos três transições que sejam sincronizadas pelo "rendez-vous" como representado na Fig. 12 abaixo.

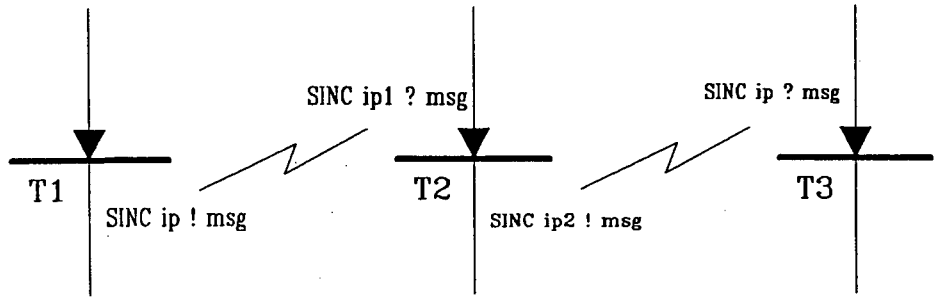


Fig. 12. Fusão de Transições com mais de uma cláusula de sincronização.

A semântica proposta sugere que quando uma transição possui duas cláusulas, uma em recepção e outra em envio, como t2, a sincronização começaria pela cláusula de recepção; para tal, t1 e t2 devem estar aptas a sincronização. Quando t1 e t2 sincronizam, t1 é disparada, e t2 vai para um estado intermediário. O estado intermediário de t2 é um estado para o qual já foram selecionados os objetos nos seus lugares de entrada, mas ainda não foram executadas suas ações. Este estado é um ponto de espera de t2 por t3, para que

a segunda sincronização seja possível. Quando t2 e t3 sincronizarem, então serão completadas as ações de t2 e t3 será disparada.

O funcionamento desta sincronização dupla pode ser representada por duas transições simples, com um lugar intermediário entre elas, como apresentado a Fig. 13 abaixo.

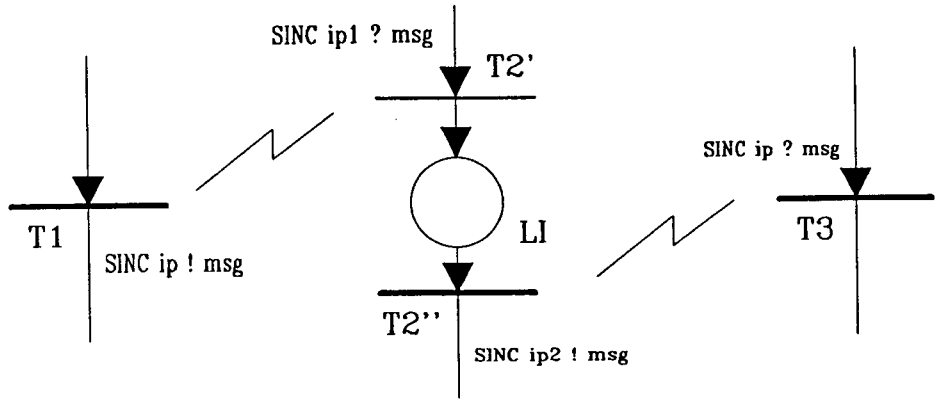


Fig. 13. Semântica da sincronização dupla.

3.3.5. Temporizações

As restrições temporais das especificações são introduzidas na linguagem através da associação à transição de uma cláusula de temporização: **DELAY** (Vmin, Vmax).

A semântica do "DELAY" é aquela definida na Rede de Petri com Temporização [Merlin, 1976] onde se associa a transição um intervalo formado por um par de numeros não negativos (Vmin, Vmax). O primeiro valor dá o tempo mínimo durante o qual a transição deve ficar sensibilizada antes de ser disparada. O segundo valor é o tempo máximo a partir do qual a transição deve disparar obrigatoriamente, desde que ela esteve continuamente sensibilizada durante o intervalo (0, Vmax).

A cláusula "DELAY" é excludente com as cláusulas de sincronização [Courtiat, 1987].

3.4. Exemplos

Dois exemplos foram escolhidos para ilustrar a linguagem, um exemplo de Automação Industrial e um exemplo de Sistemas Distribuídos que é o Protocolo Abracadabra.

3.4.1. Célula Flexível de Usinagem

Este exemplo foi apresentado na seção 2.4.2. O Apêndice 1 contém a descrição do mesmo na sintaxe da linguagem LRPO.

Observações:

1) A especificação no Apêndice 1 será utilizada como entrada para a ferramenta de simulação que será apresentada no próximo capítulo. Como esta ferramenta não suporta a noção de n-upla de objetos, as transições inicio e fim apresentam algumas condições a mais para permitir fazer a ligação entre os objetos que formam a n-upla <Pec, Pal, Maq>.

2) Ao lado das transições os símbolos P_i e J_j não fazem parte da especificação; os mesmos serão utilizados posteriormente para auxiliar na descrição do algoritmo de simulação da RPO.

3.4.2. Protocolo Abracadabra

O Protocolo Abracadabra foi especificado pela ISO como um protocolo a ser utilizado para a avaliação de Técnicas de Descrição Formal. Este protocolo se encontra descrito também em [Courtiat, 1987] e [Souza, 1989].

O serviço oferecido pelo Protocolo Abracadabra é um serviço orientado a conexão bidirecional simultânea permitindo a dois utilizadores UserA e UserB trocar, uma vez que a conexão estiver estabelecida, as Unidades de Dados de Serviço (SDU) de maneira confiável e com garantia de sequenciamento.

As primitivas de serviço utilizadas são respectivamente:

* ConReq, ConInd, ConResp e ConConf para a fase de estabelecimento de conexão;

* DatReq e DatInd para a fase de transferência de dados;

* DisReq e DisInd para a fase de liberação da conexão.

Obs: somente as primitivas DatReq e DatInd carregam um parâmetro, que é a SDU.

O Protocolo Abracadabra é simétrico, e opera entre um par de estações através de um meio "full-duplex". As Unidades de Dados de Protocolo (PDU) usadas são respectivamente:

* CR e CC para a fase de estabelecimento de conexão;

* DT e AK para a fase de tranferência de dados;

* DR e DC para a fase de liberação da conexão.

Obs: as PDU DT e AK comportam parâmetros: ambas um bit de sequenciamento e a PDU DT ainda um campo de dados.

O Protocolo Abracadabra é parametrizado por duas constantes: N (> 0) que define o número máximo de tentativas de transmissão da mesma PDU, sem a recepção do seu reconhecimento; e P que define o tempo de espera para a retransmissão.

A Fig. 14 mostra um diagrama de funcionamento do protocolo para cada uma das três fases.

A especificação completa do protocolo na linguagem LRPO, é apresentada no Apêndice 2. Para auxiliar no entendimento da modelagem, é fornecido no Apêndice 3 a forma gráfica da RPO relativa a cada fase do protocolo. A especificação considera uma configuração estática de módulos e descreve as fases de estabelecimento e de liberação da conexão Abracadabra estabelecida entre dois usuários UserA e UserB. A parte de tranferência de dados não foi representada no exemplo. A Fig. 15 ilustra a estrutura modular do protocolo Abracadabra. O módulo Meio (Medium) é representado por duas instâncias do módulo "FilaFifo", formando assim uma fila FIFO "full-duplex". Este módulo FilaFifo é representado por uma transição de inserção, por uma transição de remoção e

dois lugares. O lugar L2 conterá os objetos em transito na fila; o lugar L1 conterá um objeto especial (Fila) cujo atributo (fifo) é uma lista de inteiros que permite o controle da ordem de chegada dos elementos na fila. Algumas funções foram construídas para a manipulação da fila.

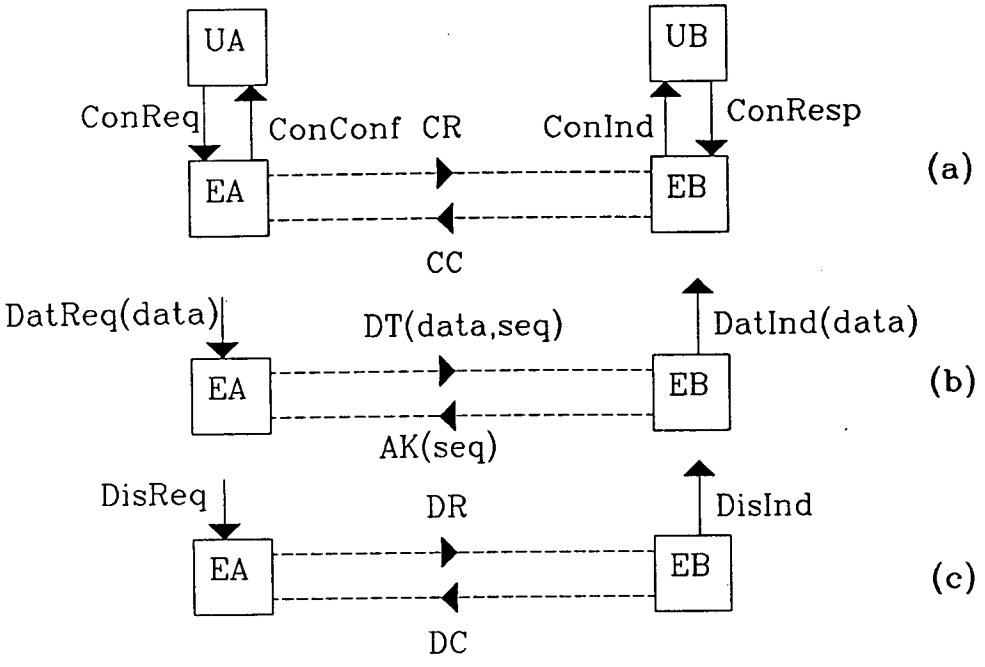


Fig. 14. Diagramas de funcionamento do protocolo Abracadabra em cada fase: a) estabelecimento de concexão; b) transferência de dados; c) liberação da conexão.

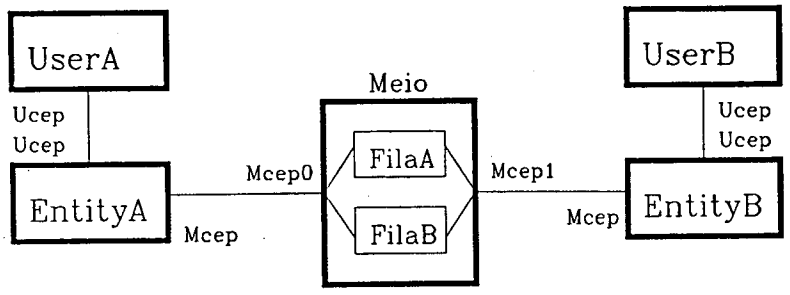


Fig. 15. Estrutura modular do protocolo Abracadabra.

3.5. Conclusões

Neste capítulo foram apresentadas as principais características de uma linguagem de descrição formal para aplicações na Automação Industrial e Sistemas Distribuídos, baseada no formalismo RPO. A linguagem se encontra hoje na forma de uma versão preliminar cuja sintaxe e semântica informal foram apresentadas anteriormente, testadas e avaliadas nas aplicações citadas.

Da avaliação da versão preliminar da linguagem LRPO, surgem proposições para nova versão desta. No que diz respeito ao aumento do nível de abstração da linguagem, trabalhos posteriores deverão analisar a introdução de algumas extensões sintáticas e semânticas, como por exemplo: a declaração de canais de comunicação externos a declaração do módulo, permitindo a várias instâncias de módulos instanciarem o mesmo canal (aos moldes da declaração CHANNEL da Estelle); e a possibilidade de definir grupos de portos (do tipo ARRAY encontrada na linguagem Estelle). Sobre esta questão da representação modular, nota-se uma grande identidade da linguagem proposta com a linguagem Estelle, e as extensões propostas vão no sentido de absorver ainda mais esta identidade.

Em termos de comunicação, a proposição de um mecanismo de "rendez-vous" múltiplo, possibilita associar várias cláusulas de sincronização às transições (fusão de várias transições), o que permite representar a difusão de uma mensagem em vários módulos. O "rendez-vous" múltiplo aumenta o poder de abstração da linguagem, todavia existem dificuldade na implementação automática do mesmo [Diaz, 1989].

A introdução na linguagem de primitivas possibilitando a configuração dinâmica (instanciação de módulos e estabelecimento/liberação de ligações em tempo de execução) aumentará as possibilidades de uso da linguagem.

A possibilidade de interação com o meio externo, tornando o módulo raiz aberto, é uma das características que a nova versão da linguagem deverá ter.

Estas duas últimas características citadas, que devem ser introduzidas em nova versão da linguagem, são de extrema importância, para que a linguagem de especificação LRPO tenha também algumas características próximas da implementação.

Conforme apresentado em [Farines, 1989a], esta linguagem faz parte de um ambiente de desenvolvimento de sistemas baseado no formalismo Rede de Petri. Este ambiente é formado por ferramentas de edição, análise, avaliação de desempenho, simulação e prototipagem, e permitirá integrar as ferramentas já disponíveis para a análise e simulação de RdP clássicas [Farines, 1989b], com as ferramentas desenvolvidas para a RPO. No próximo capítulo, será apresentado uma destas ferramentas: o simulador de RPO.

CAPÍTULO 4

SIMULADOR DE REDE DE PETRI A OBJETOS (SRPO)

4.1. Introdução

Uma vez concluída a etapa de especificação, as etapas seguintes do desenvolvimento de um sistema consistem em validar as especificações, do ponto de vista da correção e do comportamento, em avaliar o desempenho destas e, em implementá-las.

No caso de utilizar como modelo a Rede de Petri a Objetos (RPO), a especificação pode ser validada usando ferramentas baseadas nas técnicas de análise das Redes de Petri clássicas, extendidas para a classe das RPO (como: geração do grafo de estados, procura dos invariantes para cada objeto etc), bem como através da simulação. Pela simulação, é possível verificar o comportamento dinâmico do sistema percorrendo partes de seu grafo de estado, procurando comportamentos inadequados e analizando a ocorrência de seqüências de eventos. A simulação, consistindo em executar as especificações, pode ter o mesmo mecanismo de execução utilizado para a prototipagem e implementação destas.

Neste capítulo será apresentado um simulador para sistemas representados por RPO. Destaca-se em particular o desenvovimento de um mecanismo versátil e eficiente para execução do modelo, que possa ser utilizado não só para a simulação, mas também para a prototipagem e para a construção de um suporte para a implementação das especificações. Inicialmente, serão discutidas as duas abordagens normalmente utilizadas para a execução de Redes de Petri - a tradução e a interpretação - apresentando as razões da escolha da abordagem interpretativa utilizada para o simulador desenvolvido. O mecanismo de execução utilizado, tem como ponto de partida a similaridade da Rede de Petri com um Sistema de Produção (SP) de primeira ordem, que permite utilizar os conceitos e os algoritmos desse tipo de sistema para a interpretação das especificações.

Neste trabalho será apresentado um algoritmo eficiente, baseado no princípio de compilação das regras. Finalmente, serão também discutidas algumas das características que o algoritmo proposto oferece para facilitar a análise das especificações e para possibilitar a implementação em ambientes distribuídos.

4.2. Formas de Execução de um Modelo de Rede de Petri

Duas abordagens são normalmente propostas para a execução de especificações modeladas por Rede de Petri: a tradução e a interpretação [Valette, 1986].

A abordagem baseada na tradução, consiste em transformar a especificação modelada por Rede de Petri numa linguagem alvo, comumente uma linguagem procedural, que será posteriormente compilada num código executável.

A abordagem baseada na interpretação utiliza um mecanismo especializado, independente da especificação, que consiste em procurar a cada passo de execução as transições sensibilizadas e em disparar uma delas, seguindo uma estratégia pré-definida; desta forma, a marcação da rede é atualizada e pode-se começar um novo ciclo de execução.

Um argumento favorável a abordagem por tradução no caso da Rede de Petri clássica, reside na produção de um código executável eficiente, obtido a partir da tradução dos elementos da estrutura de controle da RdP em instruções básicas do código alvo. Este procedimento pode ser facilmente automatizado. Todavia a tradução da Rede de Petri a Objetos perde em eficiência por causa da simultaneidade da presença de vários objetos de classes diferentes na mesma estrutura de controle. O procedimento de tradução da RPO inicia com a determinação no modelo global de sub-redes que representam as diversas Classes de Tarefas existentes no programa resultante e da comunicação entre elas. A instanciação para os diversos objetos destas Classes de Tarefas acresce o código resultante e aumenta o número de comunicações possíveis entre instâncias. A procura das comunicações entre tarefas possíveis de serem executadas, gera um "overhead" em tempo de execução que deteriora o desempenho da implementação obtida a partir desta

abordagem. Outrossim, a geração do código é dificilmente automatizável, pois envolve decisões baseadas em heurísticas durante o processo de tradução. Uma limitação adicional desta abordagem reside no fato de que o código resultante depende da especificidade de cada rede, tornando necessária recompilações a cada mudança nas especificações [Pereira, 1990; Cantú, 1990].

Na abordagem interpretativa, o mecanismo de execução é independente da especificação, não sendo necessário uma recompilação do sistema após cada mudança nas especificações, o que pode ser útil para a prototipagem e no caso de mudança dinâmica de parte da especificação (configuração dinâmica). A principal limitação desta abordagem está na sua lentidão, relacionada em particular com a operação de procura das transições sensibilizadas.

As limitações associadas ao procedimento de tradução da RPO como a dificuldade de automatização, a necessidade de recompilação e o "overhead" em tempo de execução, pesam negativamente na escolha desta abordagem. Por outro lado, a simplicidade da abordagem interpretativa e a sua independência em relação as especificações, contribuem no sentido da escolha da mesma como mecanismo de execução da RPO. Cabe salientar também, que a abordagem interpretativa consiste na execução direta das especificações validadas, o que elimina eventuais erros introduzidos na translação da especificação para um código executável. Entretanto, é necessário analisar e propor soluções para o problema da lentidão do mecanismo de execução. A seguir será apresentado um simulador para especificações em Rede de Petri a Objetos (SRPO), baseado na abordagem interpretativa e, em seguida será descrito o algorítmo que foi utilizado pelo simulador para melhorar a eficiência desta abordagem.

4.3. Simulador de Rede de Petri a Objetos (SRPO)

4.3.1. Descrição do Protótipo do Simulador

Uma versão do protótipo do Simulador de Rede de Petri a Objetos (SRPO) foi implementada sobre um interpretador LISP (LeLisp) [Chailloux, 1985], num ambiente PC-DOS.

O simulador desenvolvido visa estudar o comportamento dinâmico do sistema modelado e depurá-lo. Para tal, ele apresenta opções de evolução passo-a-passo (onde o usuário pode escolher as transições a disparar) e automática (o que é equivalente ao caso da implementação real); além disto, este protótipo possibilita também a execução de "backtracking" (volta a trás) e a memorização de estados de interesse permitindo o retorno a este estado. O usuário terá acesso interativamente às informações correntes (marcação, caminho percorrido, etc) podendo visualizá-las ou modificá-las.

O simulador é formado por três módulos básicos: o compilador, o interpretador e a interface do sistema.

O compilador permite transformar as especificações escritas segundo a sintaxe da linguagem LRPO, numa estrutura que possa ser utilizada pelo interpretador. Esta especificação deve ser editada em um "editor standart" para ser posteriormente compilada.

O interpretador controla a execução das transições conforme as três etapas a seguir: 1) determinação das transições sensibilizadas; 2) escolha de uma delas para o disparo através de uma estratégia pré-definida e 3) disparo da transição, levando a uma nova marcação da rede. A descrição completa e detalhada do mecanismo de execução do interpretador será apresentada na seção 4.4., onde serão descritos também as técnicas utilizadas para melhorar a eficiência na execução do mesmo. No que diz respeito a estratégia de escolha das transições para o disparo, a mesma pode ser aleatória ou com a atribuição de prioridades as transições.

A interface permite ao usuário dirigir externamente a simulação através de comandos interativos e obter os resultados relativos a simulação efetuada. O QUADRO 2 fornece uma listagem dos comandos que são oferecidos pela interface do simulador, sendo os mesmos detalhados abaixo.

A seguir as funcionalidades da interface são descritas e a utilização dos comandos do simulador é ilustrada a partir do exemplo da Célula Flexível de Usinagem apresentado no Apêndice 1. Nos comandos de simulação apresentados como exemplo, os caracteres em negrito correspondem as solicitações e respostas digitadas pelo usuário.

i) Carga de arquivo para simulação:

Permite carregar o arquivo que contém o modelo especificado na linguagem LRPO para a simulação.

ex:

```
rpo> (load cfu)
* Carregada rede: cfu
```

ii) Acesso ao estado atual:

Permite a visualização da marcação da rede.

ex:

```
rpo> (m)
* Marcacao Corrente (profundidade = 0):
P1
        $8 : (Peca ^id 2 ^job (m2 m1))
        $7 : (Peca ^id 1 ^job (m1 m2 m1))
P3
        $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
        $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
P2
        $4 : (Pallet ^posicao 4 ^estado nil)
        $3 : (Pallet ^posicao 3 ^estado nil)
        $2 : (Pallet ^posicao 2 ^estado nil)
        $1 : (Pallet ^posicao 1 ^estado nil)
```

QUADRO 2. Comandos Disponíveis no Simulador **SRPO**

Auxílio:
(help) : Lista comandos disponíveis

Carga:
(load <arq>) : Carrega o arquivo **arq.ll** com a RPO para simulação

Simulação:
(s) : Mostra simplesmente as transições sensibilizadas
(d) : Disparo no modo passo-a-passo de transições
(run <n>) : Disparo automático de **n** transições
(bt <k>) : "Back-track" a profundidade **k**
(bt -<n>) : "Back-track" **n** passos de simulação
(memory) : Permite memorizar/voltar para estado de interesse

Visualização de estado:
(m) : Fornece a marcação atual

Caminho:
(path) : Fornece o caminho desde a marcação inicial

Modificação de estado:
(make <Class ^atr val ...>) : Cria objeto em um lugar da rede
(del <$n>) : Elimina o objeto identificado por **$n** da marcação
(modify <$n> ^atr val)** : Modifica valor de um atributo do objeto identificado por **$n
(init) : Restitui a marcação inicial
(start) : Reinicializa a simulação e limpa memórias

Modularidade:
(channel) : Mostra os canais de comunicação abertos entre módulos
(connect/disconnect) : Permite conectar/disconectar portos de módulos

Comandos de controle:
(prior) : Habilita/desabilita prioridades ("defaut" ())

Comandos sistema:
(system) : Acesso aos serviços do Sistema Operacional (DOS)
(lisp) : Execução de função LISP dentro do ambiente SRPO
(log/unlog) : Redirecionamento da saída para um arquivo
(quit) : Fim de simulação

---

iii) Disparo de transições:

O disparo das transições pode ser feito de dois modos:

. Modo passo-a-passo:

Possibilita ao usuário explorar partes do grafo de estados seguindo cada passo. O simulador apresenta as transições sensibilizadas na marcação corrente e o usuário escolhe uma delas para o disparo.

. Modo automático:

Usando este modo, o usuário deve fornecer um número de disparos de transições a serem efetuados, antes do controle da simulação lhe ser devolvido. Durante a execução automática o simulador seleciona de forma aleatória as transições para o disparo. Em caso de bloqueamento o usuário é avisado.

Para controlar a seleção das transições no modo automático, o usuário pode eventualmente utilizar prioridades associadas as transições, visando privilegiar alguma parte da especificação durante a simulação.

ex:

```
rpo> (run 3)
3
rpo> (d)
* Transicoes Sensibilizadas
1 : entrada : ($8 $4)   {0}
2 : fim : ($7 $1 $5)   {0}
Entre sua escolha ou 0 se nao deseja disparos: 2
  Disparada: fim : ($7 $1 $5)
```

No exemplo anterior apresenta-se o disparo de três transições no modo automático e em seguida de uma transição no modo passo-a-passo, a partir da marcação inicial. Ao lado do nome de cada transição sensibilizada são apresentados entre parênteses os objetos que a instanciam e entre chaves a prioridade associada a esta transição.

iv) Apresentação do caminho:

Apresenta-se o caminho (sequência de disparo) desde a marcação inicial até o estado atual.

ex:

```
rpo> (path)
* Caminho desde profundidade 0
  0 : ((entrada ($7 $1)))  : 1
  1 : ((rotacao ($1 $2 $3 $4)))  : 2
  2 : ((inicio ($7 $1 $5)))  : 3
  3 : ((fim ($7 $1 $5)))  : 4
```

v) Exploração o grafo de estados:

O simulador oferece a possibilidade de retornar a passos anteriores da simulação ("back-track"). Dois tipos de "back-track" são possíveis:

. "Back-track" a profundidade k:

Permite retornar a profundidade especificada. Nesta profundidade o simulador anota as transições que já foram disparadas, de modo a auxiliar o usuário a não percorrer um caminho que já tenha sido percorrido.

. "Back-track" a n passos:

Permite retornar o número de passos de simulação especificado como parâmetro. Neste caso, conta como passo de simulação os disparos, ou qualquer utilização dos comandos de modificação da marcação (make, del, modify).

ex:

```
rpo> (bt 2)
2
rpo> (s)
* Transicoes Sensibilizadas
1 : inicio : ($7 $1 $5) + {0}
2 : entrada : ($8 $4)   {0}
3 : rotacao : ($4 $1 $2 $3)   {1}
```

Após o "back-track" a simulação voltou a profundidade 2. Nesta profundidade, quando pedimos a visualização das transições sensibilizadas, o sinal "+" ao lado da transição indica ao usuário que a mesma já foi disparada anteriormente.

vi) Memorização de estados:

Permite memorizar estados de interesse, sendo possível posteriormente retornar a qualquer dos estados marcados.

ex:

```
rpo> (memory)
* Memoriza estado de interesse:
 1 - Visualizar
 2 - Memorizar
 3 - Voltar
Entre com sua escolha: 2
 Qual o nome do estado: mem1
```

vii) Modificação da marcação:

Permite alterar a marcação corrente através da criação de objetos, destruição de objetos ou modificação dos atributos de objetos já existentes nos lugares da rede.

ex:

```
rpo> (make P1 (Peca ^ident 3 ^job (M2 M2 M1)))
rpo> (del $8)
rpo> (modify $7 ^ident 4)
```

viii) Comandos de controle:

Oferecem ao usuário as seguintes funcionalidades: ativar/desativar as prioridades; acessar o Sistema Operacional (DOS); executar funções LISP; redirecionar a saída para um arquivo e terminar a sessão de simulação.

4.3.2. Simulação de um Sistema Distribuído Representado pela Linguagem LRPO

A seção anterior apresentou um simulador para sistemas implementáveis em um ambiente centralizado. A simulação de sistemas distribuídos também foi estudada, levando a introduzir modificações para adaptar o interpretador para este caso.

Para que implementações distribuídas possam ser simuladas no ambiente de simulação centralizado, descrito anteriormente, é necessário que seja introduzido a nível do interpretador do simulador apresentado, um mecanismo que permita a fusão de transições em tempo de execução.

A cada ciclo de execução, após a etapa de determinação das transições sensibilizadas pelo interpretador, determinam-se, dentro do conjunto global de transições

sensibilizadas gerado, as transições internas sensibilizadas e os pares de transições com cláusula de sincronização que estão aptos para se comunicar. Esta separação é realizada da seguinte forma: se a transisão sensibilizada possui cláusula de sincronização, então verifica-se, com o auxílio de uma lista das ligações existentes entre os módulos, se a transição correspondente está também sensibilizada. Para o disparo é escolhida de forma aleatória uma transição ou um par; no caso de um par ser disparado, as mesmas são executadas encadeadas, com troca de parâmetros entre elas. A lista de ligações entre os módulos estabelecida inicialmente pode ser modificada em curso de execução, posibilitando uma forma de configuração dinâmica da estrutura modular.

Apresenta-se a seguir o pseudo-código do algoritmo utilizado pelo interpretador para disparar as transições.

```
início
    enquanto Não chegar ao fim da simulação faça
        Seleciona as transições sensibilizadas (etapa 1);
        Separe este conjunto em:
            T := Lista das transições sensibilizadas sem cláusulas de sincronização;
            T* := Lista das transições sensibilizadas com cláusula de sincronização definida;
        se (vazio T) e (vazio T*) então Bloqueamento
        senão
            se (não vazio T) então
                Mostra transições disparáveis fim-se;
            se (não vazio T*) então
                Mostra possíveis "rendez-vous" fim-se;
            Seleciona aleatóriamente uma transição simples ou um par para o disparo (etapa 2);
            Dispara a transição (par) correspondente (etapa 3);
        fim-se;
    fim-enquanto;
fim
onde:
    etapa 1: determinação das transições sensibilizadas;
    etapa 2: seleção de uma transição para o disparo;
    etapa 3: disparo com a correspondente atualização da marcação;
correspondem as etapas do interpretador no caso centralizado.
```

Desta forma, todos os casos que se apresentam em implementações distribuídas, inclusive a introdução e retirada de módulos, podem ser simulados.

4.3.3. Exemplos de Utilização do Simulador

O Apêndice 4 apresenta parte das simulações relativas aos dois exemplos apresentados neste trabalho, que são a Célula Flexível descrita na seção 2.4.2 e o Protocolo Abracadabra descrito em 3.4.2.

No exemplo da Célula Flexível de Usinagem são associadas prioridades às transições, dando prioridade maior a todas as transições em relação transição rotacao, evitando que a mesa gire se houver alguma outra operação que possa ser efetuada. Através de simulações exaustivas, verificou-se que este critério é bastante eficaz.

Destaca-se que o exemplo do Protocolo Abracadabra serve para ilustrar a fusão de transições entre os módulos que compõem a especificação.

4.4. Algoritmo de Interpretação para a RPO

Nesta seção será apresentado o algoritmo de interpretação utilizado para a execução da RPO. O algoritmo está baseado na similaridade das Redes de Petri com os Sistemas de Produção (SP) e utiliza técnicas de compilação de regras para melhorar sua eficiência.

4.4.1. Sistemas de Produção (SP) e as Redes de Petri

Como foi apresentado em [Valette, 1986] a Rede de Petri tem um paradigma equivalente ao dos Sistemas de Produção (SP) (também chamados Sistemas de Regras de Produção), que permitem a representação do conhecimento na forma de regras.

Um SP consiste de um conjunto não-ordenado de regras de produção da forma condição/ação (chamado *Memória de Regras*); de uma base de dados representando o estado do sistema (geralmente chamada *Memória de Trabalho*) e de um interpretador (ou *Motor de Inferência*) que controla a execução das regras conforme as três etapas a seguir:

(1) *Filtragem*: determina o conjunto de todas as n-uplas de elementos da Memória de Trabalho que instanciam a parte de condição (lado esquerdo) de cada regra, tornando-a válida;

(2) *Resolução dos Conflitos*: escolhe entre as instâncias das regras válidas uma regra para o disparo, usando uma estratégia pré-definida, chamada Critério de Resolução dos Conflitos;

(3) *Ação*: avalia a parte de ação (lado direito) da regra, o que leva a atribuir uma nova semântica aos elementos capturados na parte de condição da regra selecionada ou a criar novos elementos.

A RPO consiste de um conjunto de transições da forma pré-condição/pós-condição, relacionadas entre si pela estrutura de controle da rede, formada por transições e lugares; e de um conjunto de dados representados por objetos que permite representar o estado do sistema, chamado marcação. Os estados da RPO evoluem segundo uma regra de disparo para as transições, com as etapas de determinação das transições sensibilizadas, seleção de uma delas para o disparo e finalmente o disparo com a atualização da marcação.

A partir da analogia destes dois modelos, pode-se representar uma especificação em RPO por um SP equivalente. Neste caso, cada objeto constituindo a marcação da RPO é representado por um elemento da Memória de Trabalho; a cada transição da RPO associa-se uma única regra de produção; e o mecanismo de evolução da marcação da RPO fica a cargo do interpretador de regras.

Representando a RPO desta forma, perde-se a estrutura de controle da rede. Contudo, conforme veremos no decorrer deste capítulo, o algoritmo proposto permite recuperar facilmente esta estrutura e utilizá-la de forma a melhorar o desempenho do interpretador.

Para ilustrar como uma especificação em RPO pode ser traduzida para um Sistema de Regras, vamos retomar a especificação em RPO da Célula Flexível de Usinagem encontrada no Apêndice 1. No exemplo, a transição entrada, é especificada da seguinte forma:

```
(TR entrada
      PRE   P1 (<Pec>)                    (P1)
            P2 (<Pal>)                    (P2)
            ^job(Pec) < > 0
            ^posição(Pal) = 1
            ^estado(Pal) = nil
      POS
            P2 (<Pal Pec>)
            ^estado(Pal) = ^ident(Pec) ).
```

Esta transição pode ser entendida como uma regra do tipo:

SE
uma Peça estiver no lugar P1;
um Pallet estiver no lugar P2;
o job da Peça for diferente de vazio;
a posição do Pallet for igual a constante 1;
o estado do Pallet for igual a nil;
ENTÃO
o Pallet e a Peça vão para o lugar P2;
o estado tado do Pallet é afetado com o identificador da Peça .

A transição (regra) acima manipula as variáveis Pal e Pec, que devem ser instanciadas respectivamente por objetos da classe Pallet e Peça, existentes na marcação corrente. O conjunto de requisitos que cada objeto deve instanciar (classe e atributos mencionados na parte de condições de cada regra, e no lado direito da parte de ações da mesma) formam um padrão de instanciação individual de cada variável. Estes padrões devem estar todos instanciados para que a transição seja sensibilizada (os padrões correspondentes a cada variável estão representados por P_i ao lado das transições no Apêndice 1). Além disto, se a transição contiver testes entre os atributos dos objetos, os mesmos devem ser verificadas para a sensibilização da transição. Estes testes ou ligações entre os atributos são chamados junções (e para ilustração, estão representados por J_j ao lado das transições no Apêndice 1).

4.4.2. Eficiência Computacional do Mecanismo de Evolução

Utilizando-se esta abordagem, o mecanismo de evolução da RPO baseia-se na interpretação do conjunto de regras equivalente, que permite executar as regras aplicáveis

(ou disparar as transições sensibilizadas). Neste processo, a filtragem, que consiste em determinar as regras aplicáveis (ou de forna análoga das transições sensibilizadas), é considerada uma operação de complexidade elevada que pode ocupar 90% ou mais de cada ciclo de inferência e é responsável pelo baixo desempenho dos SP. Várias soluções foram propostas para reduzir o custo desta operação, como o algoritmo RETE proposto por Forgy [1982] (base da maioria dos interpretadores da linguagem OPS5). O objetivo dos algoritmos desta família consiste em diminuir a complexidade da filtragem por eliminação das ocorrências de testes a avaliar na determinação das regras válidas. Estas ocorrências podem ser estruturais (ocorrência do mesmo predicado em duas regras diferentes que conseqüentemente só precisa ser avaliado uma única vez sobre um determinado dado) e temporais (uma regra instanciada por uma parte inalterada da base de dados não precisa ser testada novamente com estes dados).

O primeiro tipo de redundância leva a construir uma estrutura de testes otimizada equivalente a parte de condição das regras. Esta fase, que leva o nome de compilação das regras, é realizada em tempo de compilação e fornece geralmente uma rede de testes em saída a partir do conjunto de regras em entrada. O segundo tipo de ocorrência leva a definir um mecanismo incremental de propagação dos dados que só considera a parte da base de dados alterada para determinar o novo conjunto de conflitos.

Inspirado nestas considerações, foi desenvolvido no LCMI um algoritmo de filtragem baseado no princípio de compilação das regras [Garnousset, 1989]. Este compilador fornece em saída dois tipos de estruturas:

- uma rede de unificação, onde são testadas as condições para instanciação individual de cada padrão e na qual são memorizados os objetos que os instanciam;
- uma rede de junção onde são avaliadas as dependências entre os padrões, permitindo obter as instâncias das regras.

Em tempo de execução, os dados modificados são propagados na rede de unificação a fim de determinar as instâncias de padrões alteradas. O conjunto dessas alterações

permite determinar, a seguir, as alterações sobre as instâncias das regras na rede de junção.

Outras características introduzidas no algoritmo permitem melhorar ainda mais a eficiência do interpretador, como será apresentado em detalhes a seguir, na descrição do algoritmo de filtragem.

4.4.3. Descrição do Algoritmo de Filtragem do Interpretador

O interpretador utilizado para a interpretação da RPO foi desenvolvido no LCMI e sua primeira versão foi construída para aplicações em Sistemas Especialistas [Garnousset, 1988; Kaestner, 1989]. A segunda versão, apresentada em [Garnousset, 1989] apresenta um algoritmo mais eficiente para a interpretação das regras e trata com maior detalhes sua aplicação para a execução da RPO.

i) A Rede de Unificação

A rede de unificação gerada pelo compilador é construída de forma incremental a partir do conjunto de transições, adicionando a cada nova transição todas as características do conjunto de padrões necessários para a instanciação da mesma.

Para exemplificar, iniciando-se a compilação das transições do exemplo da Célula Flexível de Usinagem visto anteriormente, a Fig. 16a mostra a estrutura gerada a partir da compilação da primeira linha da transição entrada, que corresponde a existência de um objeto da Classe Peça no lugar P1. Prosseguindo a compilação de maneira incremental, a Fig. 16b mostra a configuração da rede de unificação após a compilação da segunda linha da transição entrada . A Fig. 16c mostra a rede após a compilação da parte restante da transição entrada , composta por condições sobre duas classes de objetos, formando os padrões P_1 e P_2. Para o padrão P_1 é necessário que exista um objeto da classe Peca, com os seguintes atributos definidos: identificador (^ident), job (^job) diferente de vazio, e que esteja no lugar (^lugar) P1. Para o padrão *P2* é necessário que exista um objeto da classe

Pallet, com os seguintes atributos definidos: posição (^posição) igual a 1, estado (^estado) igual a vazio, e que esteja no lugar (^lugar) P2. A Fig. 16d mostra a rede de unificação completa obtida após a compilação de todas as regras deste exemplo.

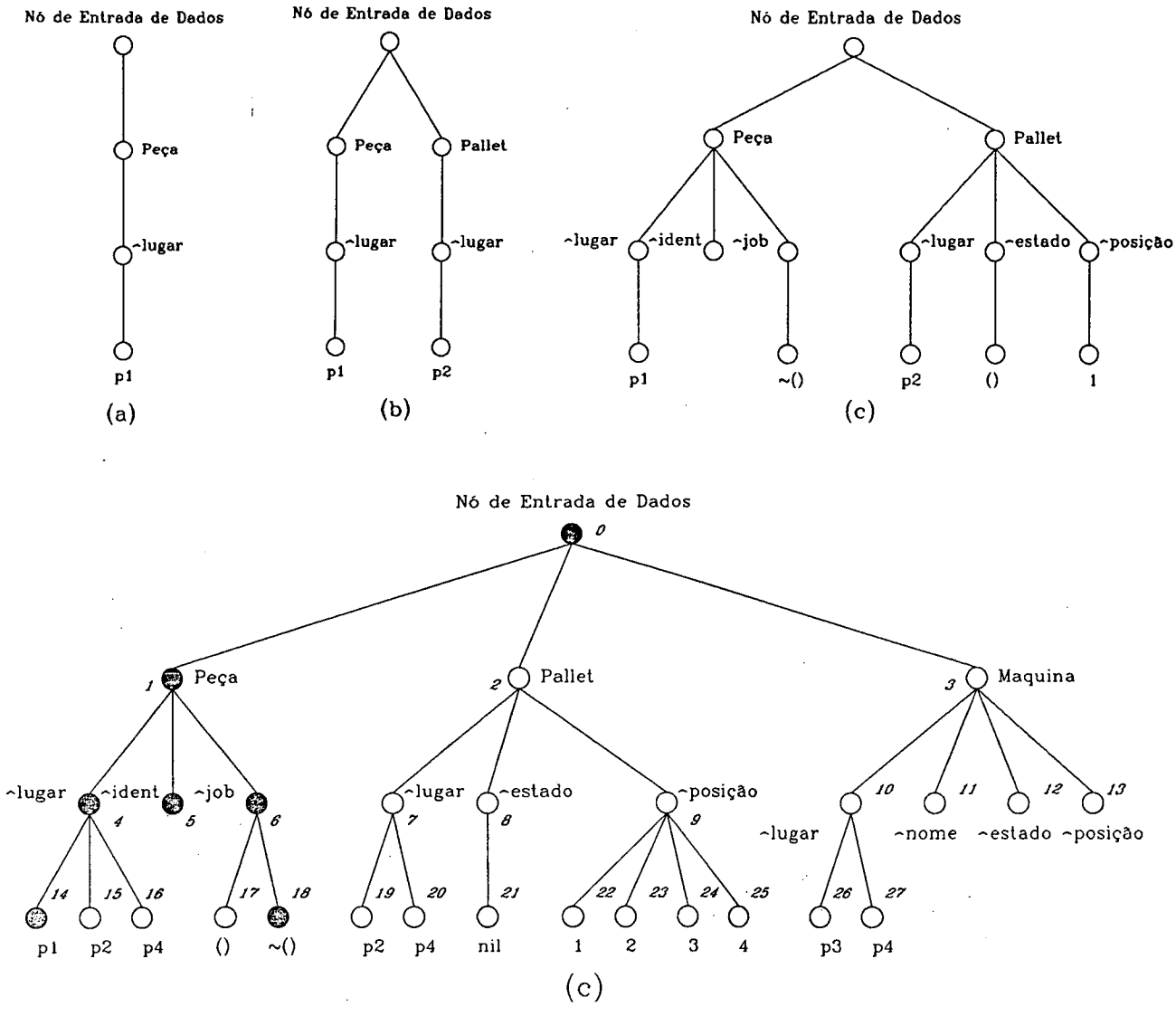


Fig. 16. Rede de Unificação: a-b-c) construção incremental da rede; d) rede global para o exemplo da CFU.

Os testes para a instanciação dos padrões na rede de unificação é realizado por sub-testes em diferentes ramos. Por exemplo, o objeto da marcação inicial contido no lugar P1, etiquetado por

$7 : (Peça ^id 1 ^job (M1 M2 M3))

quando propagado na rede, instancia o conjunto de nós {1,4,5,6,14,18}, escurecidos na Fig. 16d. Nota-se na figura que o lugar onde está o objeto na rede é representado como um atributo deste objeto, no caso o nó 14 da rede. A Fig. 17 mostra o resultado da propagação de todos os objetos da marcação inicial do exemplo dado no Apêndice 1 através da rede de unificação.

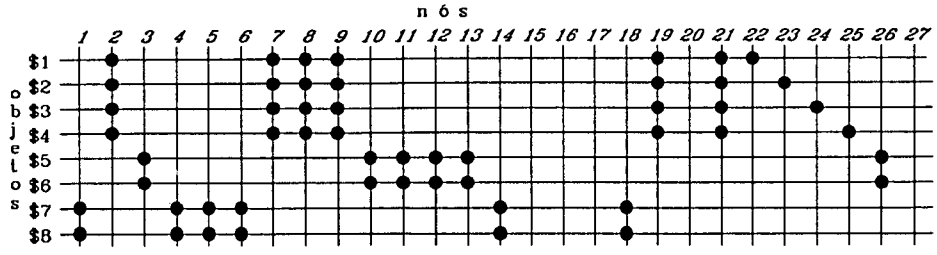


Fig. 17. Nós instanciados na marcação inicial.

A partir do conjunto de transições dado, o compilador extrai 14 padrões $P_1,...,P_{14}$ conforme ilustrado ao lado das transições na especificação contida no Apêndice 1. Como no exemplo apresentado nenhuma transição possui um padrão comum com outra transição, cada variável de cada regra corresponde a exatamente um padrão, sendo cada um deles definido por um conjunto de nós da rede de unificação, que devem todos serem instanciados pelo mesmo objeto, para que o padrão possa ser instanciado. O conjunto de nós que formam cada padrão é dado a seguir.

P_1 = {14,5,18}
P_2 = {19,22,21}
P_3 = {15,5,17}
P_4 = {19,8,25}
P_5 = {19,22}
P_5 = {19,23}
P_6 = {19,24}
P_8 = {19,25}
P_9 = {15,5,6}
P_{10} = {19,8,9}
P_{11} = {26,11,13}
P_{12} = {16,5}
P_{13} = {20,8}
P_{14} = {27,12}

Em tempo de execução, os dados modificados da Memória de Trabalho são propagados na rede de unificação, a fim de determinar as instâncias de padrões alterados. A verificação da instanciação de cada padrão é realizada por um teste binário, definido como segue. A cada objeto na Memória de Trabalho é associado uma cadeia binária,

limitada pelo número de nós da sub-árvore correspondente ao objeto. O n-ésimo componente da cadeia é setado para 1 se o objeto instancia o nó correspondente, e 0 se não instancia. Do mesmo modo, cada padrão é representado por uma cadeia binária, onde somente os nós que formam o padrão são setados para 1. Assim sendo, através de uma simples operação binária pode se verificar se o padrão é instanciado pelo objeto, como ilustra a Fig. 18, onde o padrão P_1 é instanciado pelo objeto $7.

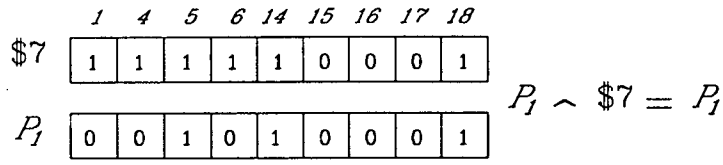


Fig. 18. Operação binária para a instanciação dos padrões.

Uma vez determinado todas as instâncias dos padrões que formam as regras, todas as junções entre os padrões devem ser testadas, de modo a determinar as transições sensibilizadas.

ii) A Rede de Junção

A operação de junção consiste em testar em tempo de execução todas as ligações existentes entre os atributos dos objetos que estão instanciando os respectivos padrões da mesma regra. Por exemplo, a junção entre os padrões que formam a transição entrada não exige nenhuma operação especial entre os mesmos, sendo necessário apenas que ambos estejam instanciados ao mesmo tempo. A Fig. 19a ilustra este caso, onde os retângulos representam memórias intermediárias que mantém o resultado da junção, sendo cada resultado obtido pelo produto cartesiano de todos os conjuntos de instâncias de padrões. O símbolo X representa o produto cartesiano entre os padrões, que é armazenado no retângulo de baixo da Fig. 19a.

No caso da transição inicio, existe as seguintes operações de junções entre seus padrões P_9, P_{10} e P_{11}, correspondentes respectivamente as variáveis Pal, Pec e Maq:

^posição(Pal) = ^posição(Maq) (J_1)
^estado(Pal) = ^ident(Pec) (J_2)
^nome(Maq) = (primeira ^job(Pec)) (J_3)

A Fig. 19d mostra como as junções podem ser testadas para esta transição. O símbolo $\bowtie_j$ representa a operação de junção entre os padrões. Neste caso, para que a junção seja satisfeita é necessário que o predicado entre os dois nós de entrada na junção sejam verificados.

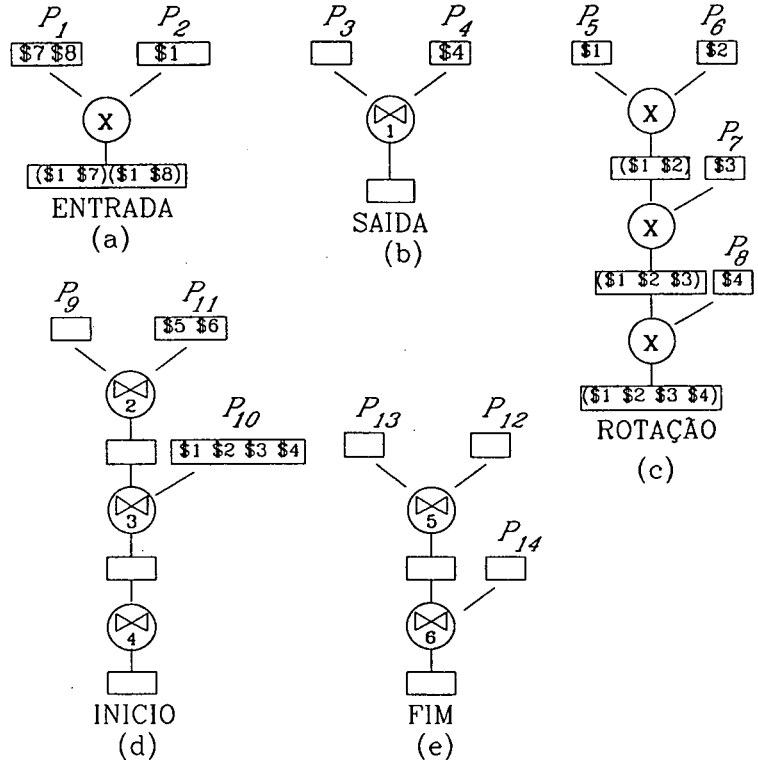


Fig. 19. Rede de junções para cada transição.

O problema do teste das junções é resolvido construindo durante a compilação uma rede de junção para cada regra, sem considerar a existência de junções comuns entre duas regras. Esta característica permite que a rede de junção seja construída de forma incremental, a cada nova regra tratada pelo compilador.

Para otimizar as junções relativas a uma regra, foi utilizada uma heurística simples, que permite reduzir o montante de dados intermediários que devem ser testados pelas

operações de junção. Esta heurística consiste em ordenar as operações de junção de acordo com o predicado que lhe é associado. Primeiramente é testado a igualdade, por ser o mais restritivo, e em seguida os outros predicados, seguindo a ordem: >=, <=, >, <, <>. Todas os demais predicados são ordenados conforme a ordem em que são escritos na parte antecedente da regra. Na Fig. 19 são apresentados todas as operações de junções sobre o conjunto das transições que formam o exemplo do Apêndice 1. Na mesma figura é também possível verificar o resultado da operação de junção obtida quando a marcação inicial é propagada. É importante notar que as transições entrada e rotacao podem ser disparadas, cada uma com qualquer das n-uplas de objetos que a instanciam: objetos ($1 $7) ou ($1 $8) para a transição entrada e ($1 $2 $3 $4) para a transição rotacao. A estratégia de resolução dos conflitos escolhida permitirá escolher uma das transições para o disparo.

Através da rede de unificação e da rede de junção apresentada, é possível obter de forma eficiente as transições sensibilizadas da especificação fornecida. Será apresentado a seguir outra característica do interpretador, que permite a compilação de algumas conclusões das regras.

iii) Compilação das Conclusões

Nesta seção será discutida a forma do algoritmo apresentado tratar as modificações dos dados na Memória de Trabalho.

Suponha que a partir da marcação inicial do exemplo apresentado, seja disparada a transição entrada com a n-upla de objetos ($1 $7) que a instancia. Neste caso o objeto

$7 : (Peça ^ident 1 ^job (M1 M2 M3))

sofre uma única modificação no seu atributo ^lugar (lugar onde o mesmo está contido na rede), que passa do valor P1 para o valor P2. A divisão dos testes para cada atributo dos objetos em diferentes ramos da rede de unificação, torna desnecessária a repropagação de todo o objeto nesta, sendo suficiente propagar a modificação ocorida.

Outrossim, no caso apresentado, mesmo esta propagação parcial pode ser evitada, antecipando o processamento da mesma em tempo de compilação. Tal processamento é possível porque o objeto assume um novo valor, que é um dado constante, independente de outros dados da Memória de Trabalho. Na Fig. 20a é possível observar as modificações que o atributo ^lugar do objeto Peca pode sofrer quando as transições são disparadas; em particular a modificação deste atributo do valor P1 a P2, quando a transição entrada é disparada é representada pelo arco orientado entre o nó 14 e o nó 15 da rede de unificação. Na Fig. 20, são apresentadas as diversas compilações das conclusões das regras do exemplo da Célula Flexível de Usinagem. Destaca-se os seguintes resultados fundamentais. O primeiro diz respeito ao ganho de eficiência do interpretador quando tratando das modificações dinâmicas dos dados na Memória de Trabalho. O segundo à aspectos relativos a análise da RPO: observa-se na sub-árvore correspondente ao atributo ^lugar de cada Classe de Objetos, que os arcos representando as modificações de um nó para outro, descrevem o caminho percorrido pelo objeto na rede através do disparo das transições, exceção feita àqueles relativos a transições neutras que não são representados na rede; estes caminhos representam a estrutura de controle da RPO, e através deles é possível determinar as componentes invariantes de lugar da RPO para cada objeto.

Em conclusão, todas as mudanças ocorridas nas instâncias dos padrões são determinadas reavaliando todas mudanças ocorridas nos nós que compõe o padrão correspondente. Os padrões que sofreram modificações são então propagados através da rede de junção, permitindo gerar as novas transições sensibilizadas.

Nesta seção, foi apresentado um interpretador para permitir a execução da RPO. O interpretador tem o comportamento semelhante ao do Mecanismo de Inferência dos Sistemas de Produção, e sua eficiência foi melhorada através da utilização de um algoritmo de compilação das regras. O caracter incremental da compilação das regras permite adicionar ou remover transições da especificação, sem necessitar uma recompilação completa da RPO; esta característica apresenta particular interesse no caso da

prototipagem. Atualmente o algoritmo de interpretação se encontra disponível numa versão em LISP interpretado. Entretanto, uma melhoria da eficiência pode ser obtida a partir da escrita do mesmo no código de uma linguagem mais eficiente (C por exemplo).

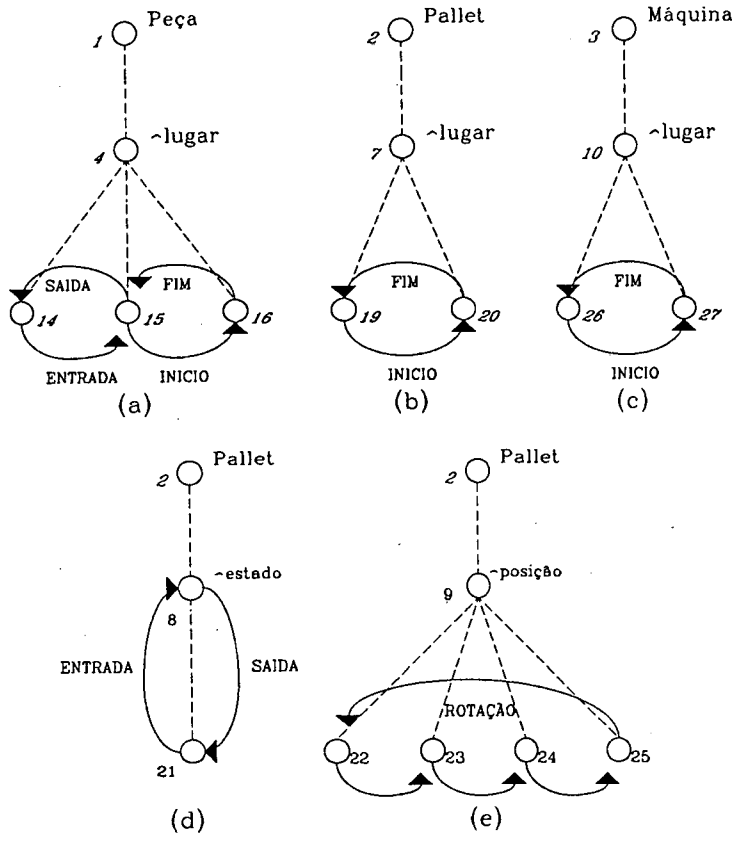


Fig. 20. Compilação das conclusões das regras.

4.4.4. Outras Características Oferecidas pelo Mecanismo de Evolução Proposto

i) Implementação em Ambientes Distribuídos

A simulação, a prototipagem e a implementação real de um sistema centralizado, segundo a abordagem interpretativa, utilizam o mesmo mecanismo de inferência. Neste caso, é equivalente apresentar o mecanismo de evolução para simulação ou para a implementação em um ambiente centralizado.

No que diz respeito a implementação em ambientes distribuídos, um mecanismo de execução similar ao previamente descrito deve ser usado em cada sítio físico que compõe o sistema. Em cada sítio, serão implementados um (ou vários) módulos especificados na linguagem LRPO citada anteriormente. As transições de comunicação entre os módulos são formadas por regras especiais que precisam receber informações enviadas pelos módulos remotos em sua Memória de Trabalho local. Escolhendo-se um mecanismo de comunicação síncrona, a partir de cada regra de envio, um chamado síncrono com a mensagem associada será enviado ao sítio remoto; as ações destas regras só serão executadas, sincronamente, quando receberem a mensagem resposta da regra de recepção do sítio remoto. Por outro lado, a habilitação das regras de recepção no sítio remoto é possível quando, além da verificação das suas condições internas, a mensagem recebida do módulo de envio está presente na sua Memória de Trabalho local.

ii) Procura dos Invariantes da RPO

O cálculo dos invariantes para a Rede de Petri clássica pode ser obtido através de métodos algébricos, aplicados sobre a Matriz de Incidência, que contém informações sobre a estrutura de controle da rede em análise.

No caso da RPO, a Matriz de Incidência se apresenta de forma simbólica. Em consequência é necessário através de um método de Projeção, obter uma matriz de incidência numérica para cada objeto distinto da rede, a partir do qual é possível obter os invariantes relativos àquele objeto.

Como apresentado na seção 4.4.3.iii, a compilação das conclusões das regras, permite obter para cada objeto todos os caminhos que o mesmo percorre na rede. A partir destes caminhos, se obtém facilmente para cada objeto uma Rede de Petri clássica, e a partir dela podemos utilizar os métodos conhecidos para determinar os invariantes de lugar. A Fig. 21 mostra as projeções encontradas, obtidas a partir da informação da compilação das conclusões vistas anteriormente. Na Fig. 21b os arcos tracejados, não são obtidos da compilação das conclusões, mas se referem a Transições Neutras, cujo disparo

não modifica o conjunto de marcações acessíveis. Estes arcos podem ser obtidos visualmente na rede, e não são releventes para a determinação dos invariantes de lugar de cada objeto.

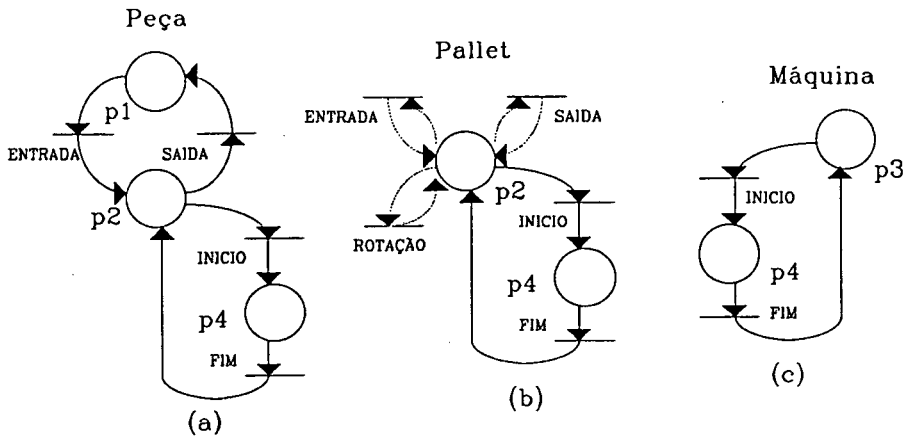


Fig. 21. Projeções para cada Objeto da RPO.

iii) Geração do Grafo de Estados

A geração do grafo de estados consiste na procura completa de todos os estados obtidos através do disparo das transições. A cada estado gerado, o algoritmo controlando a procura, deve verificar a existência ou não deste no conjunto dos estados já gerados.

A memorização de cada estado da Memória de Trabalho e a verificação, a cada passo da geração do grafo, da existência ou não do novo estado gerado, pode ser extremamente custosa. Tendo em vista estas questões, esta se investigando também a utilização das estruturas geradas pelo algoritmo para auxílio nesta tarefa, através da determinação de uma melhor estrutura e agregação dos dados para melhorar o desempenho desta procura e memorização visando a determinação do grafo de estados. A utilização de métodos de procura "hashing" também está sendo investigada.

4.5. Conclusão

Neste capítulo foi apresentado um simulador para aplicações descritas por Rede de Petri a Objetos. O simulador descrito implementa as regras de disparo das transições da RPO através de um interpretador, similar ao Mecanismo de Inferência dos Sistemas de Produção (SP). Para aumentar o desempenho do interpretador foi utilizado um algoritmo de compilação das regras, que além de permitir a execução eficiente da RPO, permite auxiliar na análise, através da obtenção das projeções de cada objeto da RPO, que permite determinar os invariantes de lugar, e na geração do grafo de estados do sistema. Este mecanismo pode ser utilizado também na implementação direta das especificações descritas por RPO no caso de uma implementação centralizada ou distribuída.

CAPÍTULO 5

CONCLUSÃO

O objetivo deste trabalho foi contribuir para a formação de um ambiente de desenvolvimento de sistemas, baseado no formalismo Rede de Petri. O ambiente proposto procura auxiliar o projetista através de uma metodologia de projeto e da utilização de ferramentas automatizadas que permitam atuar em todas as etapas do desenvolvimento do sistema.

Tendo em vista as limitações do modelo Rede de Petri no que se refere a descrição dos dados, foi utilizada a extensão Rede de Petri a Objetos (RPO), onde os dados são representados na forma de objetos do tipo relação. Foi proposta uma linguagem de descrição formal de sistemas (LRPO) com características de estruturação hierárquica e de modularidade, e que utiliza a RPO como modelo de base para a representação do comportamento e dos dados; a comunicação entre os módulos é síncrona, do tipo "rendez-vous". Um conjunto de ferramentas de simulação e prototipagem (SRPO) que permite executar as especificações descritas nesta linguagem foi também apresentado.

O simulador desenvolvido, permite executar as especificações fazendo evoluir a marcação da RPO. A abordagem utilizada para o desenvolvimento do mecanismo de evolução do simulador, é baseada na similaridade entre os Sistemas de Produção e o modelo de RPO. Para melhorar a eficiência do interpretador foram introduzidas técnicas de compilação de regras, visando diminuir o custo da etapa de filtragem que determina as transições sensibilizadas. Além da possibilidade de execução da RPO, o mecanismo permitirá também o desenvolvimento de alguns módulos de análise, através da geração do grafo de estados do sistema e da determinação das projeções para cada objeto da RPO objetivando encontrar os invariantes de lugar. Estas duas técnicas podem ser integradas na ferramenta ARP [Farines, 1989b], desenvolvida para análise e simulação das Redes de

Petri clássicas. As ferramentas de análise e sua integração no ambiente ARP são objetos de estudos e pesquisas no Laboratório de Controle e Microinformática (LCMI).

Outrossim, o mecanismo de evolução da RPO proposto, deve ser utilizado também para a implementação direta das especificações. Estudos estão sendo empreendidos em direção do uso deste mecanismo para uma implementação distribuída.

BIBLIOGRAFIA

Alanche,P.; Benzakour,K.; Dollé,F.; Gillet,P.; Rodrigues,P.; Valette,R., (1984). **"PSI: a Petri Net Based Simulator for Flexible Manufacturing Systems".** Lecture Notes in Computer Science 188, p.1-14.

Azema,P.; Nordgard,K.; Lloret,J.C., (1988). **"PROFIP: Liaison de Donneés".** Rapport LAAS/CNRS, Tousouse, France.

Azema,P.; Lloret,J.C.; Papapanagiotakis,G.; Vernadat,F., (1988). **"Estelle Validation and PROLOG Interpreted Petri Nets".** Rapport LAAS/CNRS, Toulouse, France.

Bako,B.; Valette,R., (1989). **"Systemes de Compilation de Regles et Réseuax de Petri a Objects".** Rapport LAAS/CNRS, Toulouse, France.

Benzakour,K., (1985). **"Simulateur de Command Logique et de Procédés".** These de Docteur 3º Cicle, Universite Paul Sabatier, Toulouse, France.

Billington,J.; Wheeler,G.R.; Wilbur-Wam,M.C., (1988). **"PROTEAN: A High-level Petri Net Tool for the Specification and Verification of Communication Protocols".** IEEE Transactions on Software Engineering, August:301-316.

Bougé,L., (1988). **"Semantique du Paralélisme: um Tour d'Horizon".** Rapport LIENS, N.88-6, École Normale Supérieure, Paris, France.

Bruijning,J. and SPECS Consortium, (1987). **"Evaluation and Integration of Especification Languages".** Computer Networks and ISDN Systems, Vol.13: 75-89.

Budkowski,S. ; Dembinski,P., (1987). **"An Introduction to Estelle: A Specification Languague for Distributed Systems".** Computer Networks and ISDN Systems, N.1, Vol.14: 3-23.

Cantu,E.; Farines,J-M.; Garnousset,H.E., (1990). **"Implementação de Especificações de Sistemas Descritos por Rede de Petri a Objetos".** Submetido ao 8º Congresso Brasileiro de Automática, Belém, PA.

Chailoux,J., (1985). "Le-Lisp de INRIA. Le Manuel de Référence", version 15, INRIA.

Courtiat,J-P., (1987). **"Contribution a la Description Formelle de Protocoles".** These de Docteur d'Etat, Université Paul Sabatier,Toulouse, France.

Courtiat,J-P., (1988a) **"Estelle*: A Powerful Dialect of Estelle for OSI Protocol Descriptions".** 8th Symposium on Protocol Specification, Testing and Verification, Atlantic City, USA.

Courtiat,J-P.; Diaz,M., (1988b). **"Description Formelle de Protocoles OSI em Estelle.** Rapport LAAS/CNRS, Toulouse, France.

Diaz,M., (1982). **"Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models".** 2th International Workshop on Protocol Specification, Testing and Verification, Idyllwild, CA, USA, p.465-510.

Diaz,M., (1989). **"Enviroments Logiciels pour la Conception des Protocoles dans les Systèmes Distribués".** Actes du Séminaire Franco-Bresilien sur les Systèmes Informatiques Répartis: 28-35, Florianópolis, SC.

ISO-DIS 9074, (1987). **"Estelle - A Formal Description Technique Based on an Extended State Transition Model".**

ISO-DIS 8807, (1988). **"LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour".**

Farines,J-M.; Cantú,E.; Garnousset,H.E.; Maziero,C.A., (1989a). **"ARP: Uma Ferramenta para o Desenvolvimento de Software em Aplicações Distribuídas".** Anais do Seminário Franco-Brasileiro em Sistemas Informáticos Distribuídos: 73-79, Florianópolis, SC.

Farines,J-M.; Maziero,C.A.; Franco,L.A.E., (1989b). **"O Ambiente ARP de Análise e Simulação de Sistemas Modelados por Rede de Petri".** Anáis do III Simpósio Brasileiro de Engenharia de Software, Recife, PE.

Forgy,C.L., (1982). **"RETE: A Fast Algotithm for thr Many Pattern/Many Objetct Pattern Mach Problem".** Artificial Intelligence, September, N.19: 17-37.

Gallab,M., (1985). **"Coping with Complexitu in Inference and Planning Systems".** Robotics Research, Fauferas and Giralt Eds., MIT Press, Cambridge, MA.

Garnousset,H.E.; Kaestner,C.A., (1988). **"SP1: Motor de Inferência para Sistemas de Regras de Produção".** 5º Simpósio Brasileiro de Inteligência Artificial, Natal, RN.

Garnousset,H.E.; Farines,J-M.; Cantú,E., (1989)."**Efficient Tools for Analysis and Implementation of Manufacturing Systems Modelled by Petri Nets with Objects: A Production Rules Compilation Based Approach".** 15º Conference IEEE Industrial Electronics Society - IECON'89, Vol.III: 543,549, Philadelphia, USA.

Genrich,H.J., (1987). **"Predicate/Transition Nets".** Report of Institut fur Methodiche Greendlagen, R.F.Germany.

Juanole,G.; Diaz,M.; Algayres,B.; Martin,J.M., (1985). **"Methodologie de Conception et de Specification Formelle de Couples Service-Protocole".** Rapport nº 85.260 LASS/CNRS, Toulouse, France.

Kaestner,C.A., (1989). **"Contribuição ao Estudo e Desenvolvimento de um Sistema de Regras de Produção".** Dissertação de Mestrado do DEEL/UFSC, Florianópolis, SC.

Lloret,J.C.; Azema,P.; Vernadat,F., (1988). **"Réseaux PrT Labellés: Structuration et Composition".** Rapport nº 88350 LAAS/CNRS, Toulouse, France.

Merlin,P.M.; Farber,D.J., (1976). **"Recoverability of Communications Protocols".** IEEE Trans. on Communications, September.

Papapanagiotakis,G., (1986). **"Reseaux Predicats et Logique Temporelle pour la Vérification des Systèmes Répartis".** These de Docteur-Ingenieur, Universite Paul Sabatier, Toulouse, France.

Pereira e Silva,R., (1990). **"Proposta de uma Metodologia de Tradução da Rede de Petri a Objetos".** Dissertação de Mestrado em preparação no DEEL/UFSC, Florianópolis, SC.

Peterson,J.L., (1981). "Petri Net Theory and Modeling of Systems", Prentice-Hall. USA.

Saqui-Sannes,P.de; Courtiat,J-P., (1989). **"Rapid Prototyping of an Estelle Simulator: ESTIM".** The Formal Description Tecnique Estelle, Ed. North-Holland, p.353-379.

Sibertin-Blanc,C., (1984). **"Petri Nets with Individuals or Objects Insted of Tokens".** Internal Report: 1-35, Université des Sciences Sociales, Toulouse, France.

Sibertin-Blanc,C., (1985). **"High-level Petri Nets with Data Structure".** 6th European Workshop on Applications and Theory of Petri nets, Helsinski, Finland.

Sibertin-Blanc,C., (1988). **"Le Prototypage des Applications Interactives a l'aide de Reseaux de Petri".** Rapport Interne: 837-855, Université des Sciences Sociales, UER Informatique, Toulouse, France.

Souza,W.L.de; Souza,J.N.de, (1989). **"Simulaçao de Especificações Formais de Protocolos de Comunicação".** Anais do Seminário Franco-Brasileiro em Sistemas Informáticos Distribuídos: 103-110, Florianópolis, SC.

Roux,J.L.; Juanole,G., (1987). **"Functional and Performance Analysis Using Extended Time Petri Nets".** Workshop on Petri Nets and Performance Models, p.14-23.

Valette,R., (1986). **"Nets in Production Systems".** Lecture Notes in Computer Science 255: Advance in Petri Nets, Part II: 191-217.

Valette,R., (1988). **"Les Reseaux de Petri".** Rapport LAAS/CNRS. Toulouse, France.

APÊNDICE 1

CÉLULA FLEXÍVEL DE USINAGEM

```
(STRUCTURE Célula_Flexível
   (CLASS
          (Peça ^ident ^job)
          (Máquina ^nome ^estado ^posição)
          (Pallet ^estado ^posição) )
   (VAR   Pec    Peça
          Maq    Máquina
          Pal    Pallet
          Pal'   Pallet
          Pal"   Pallet
          Pal"'  Pallet )
   (PLACE  P1  (<Peça>)
           P2  (<Pallet> <Pallet Peça>)
           P3  (<Máquina>)
           P4  (<Peça Máquina Pallet>) )
   (TR entrada 0
      PRE
         P1 (<Pec>)                          (P1)
         P2 (<Pal>)                          (P2)
         ^job(Pec) <> ()
         ^posição(Pal) = 1
         ^estado(Pal) = nil
      POS
         P2 (<Pec> <Pal>)
         ^estado(Pal) = ^ident(Pec) )
   (TR saída 0
      PRE
         P2 (<Pec> <Pal>)                    (P3 P4)
         ^job(Pec) = ()
         ^posição(Pal) = 4
         ^estado(Pal) = ^ident(Pec)          (J1)
      POS
         P1 (<Pec>)
         P2 (<Pal>)
         ^estado(Pal) = nil )
   (TR rotação 1
      PRE
         P2 (<Pal> <Pal'> <Pal"> <Pal"'>)
         ^posição(Pal) = 1                   (P5)
         ^posição(Pal') = 2                  (P6)
         ^posição(Pal") = 3                  (P7)
         ^posição(Pal"') = 4                 (P8)
      POS
         P2 (<Pal> <Pal'> <Pal"> <Pal"'>)
         ^posição(Pal) = 2
         ^posição(Pal') = 3
         ^posição(Pal") = 4
         ^posição(Pal"') = 1 )
   (TR início 0
      PRE
         P2 (<Pec> <Pal>)                    (P9P10)
         P3 (<Maq>)                          (P11)
         ^posição(Pal) = ^posição(Maq)       (J2)
         ^estado(Pal) = ^ident(Pec)          (J3)
         ^nome(Maq) = (primeiro ^job(Pec))   (J4)
      POS
         P4 (<Pec Pal Maq>)
         ^job(Pec) = (resto ^job(Pec))
         ^estado(Maq) = ^ident(Pec) )
   (TR fim 0
      PRE
         P4 (<Pec Pal Maq>)                  (P12P13P14)
         ^estado(Maq) = ^ident(Pec)          (J5)
         ^estado(Pal) = ^ident(Pec)          (J6)
      POS
         P2 (<Pec> <Pal>)
         P3 (<Maq>)
         ^estado(Maq) = nil )
   (INIT
      P2 (
         (Pallet ^posição 1 ^estado nil)
         (Pallet ^posição 2 ^estado nil)
         (Pallet ^posição 3 ^estado nil)
         (Pallet ^posição 4 ^estado nil) )
      P3 (
         (Máquina    ^nome m1
                     ^posição 2
                     ^estado nil)
         (Máquina    ^nome m2
                     ^posição 3
                     ^estado nil) )
      P1 (
         (Peça ^ident p1 ^job (m1 m2 m1))
         (Peça ^ident p2 ^job (m2 m1)) ) )
)
(de primeiro (lista) (car lista))
(de resto (lista) (cdr lista))
```

APÊNDICE 2

PROTOCOLO ABRACADABRA

```
(STRUCTURE Abracadabra
   (REGISTER
      @MTP 1
      @Epsilon 0.2
      @N 5 )
   (MODULE AbraEntity
      (REGISTER
         @N-attempts 0
         @P (* 2 (+ @MTP @Epsilon)) )
      (CLASS
         (Ficha)
         (Pdu ^code ^seq ^data)
         (Sdu ^data) )
      (VAR
         F        (Ficha)
         P        (Pdu)
         S        (Sdu) )
      (PORTS
         UCEP  IN   (ConRec ConResp
                     DisReq DatReq (S) )
               OUT  (ConInd ConConf
                     DisInd DatInd (S) )
         MCEP
               IN   (UnitInd (P) )
               OUT  (UnitReq (P) ) )
      (PLACE
         Closed          (<Ficha>)
         WFCC            (<Ficha>)
         WFUR            (<Ficha>)
         Closing         (<Ficha>)
         Error           (<Ficha>)
         Open            (<Ficha>)
         SemiClosed      (<Ficha>)
         SavingError     (<Ficha>)
         DelayWFCC       (<Ficha>)
         DelayClosing    (<Ficha>)
         PlaceWFCC       (<Pdu>)
         PlaceClosed     (<Pdu>)
         PlaceClosing    (<Pdu>) )

; FASE DE ABERTURA DE CONEXAO

; Comportamento do Iniciador da conexao

(TR T1
   PRE    Closed (<F>)
   SINC   UCEP ? ConReq
   SINC   MCEP ! UnitReq(Pdu ^code CR)
   POS    WFCC (<F>) )

(TR T2
   PRE    WFCC (<F>)
   SINC   MCEP ? UnitInd(P)
   POS    PlaceWFCC (<P>) )

; Retira mensagem da fila e coloca em lugar intermediario

(TR T3
   PRE    PlaceWFCC (<P>)
   COND ^code(P) or (= CR) (= CC)
          SINC   UCEP ! ConConf
          POS    Open (<F>)
   OR     ^code(P) = DR
          SINC   UCEP ! DisInd
          POS    SemiClosed (<F>)
   OR    ^code(P) and (<>CR) (<>CC)
                 (<>DR)
          POS  WFCC (<F>) )

(TR T6
   PRE    WFCC (<F>)
   SINC   UCEP ? DisReq
   SINC   MCEP ! UnitReq(Pdu ^code DR)
   POS    Closing (<F>) )

(TR T7
   PRE    WFCC (<F>)
   COND @N-attempts <  @N
          DELAY (@P, @P)
          POS  DelayWFCC (<F>)
          @N-attempts = (+ 1 @N-attempts)
   OR     @N-attempts > = @N
          POS  Error (<F>) )

(TR T9
   PRE    DelayWFCC (<F>)
   SINC   MCEP ! UnitReq(Pdu ^code CR)
   POS    WFCC (<F>) )
```

```
; Comportamento em resposta de pedido de
conexao

(TR T10
   PRE     Closed (<F>)
   SINC    MCEP ? UnitInd(P)
   POS     PlaceClosed (<P>) )
; Retira mensagem da fila e coloca em lugar intermediario

(TR T11
   PRE     PlaceClosed (<P>)
   COND    ^code(P) = CR
           SINC    UCEP ! ConInd
           POS     WFUR (<F>)
   OR      ^code(P) = DR
           SINC    MCEP !
                   UnitReq(Pdu ^code DC)
           POS     Closed (<F>)
   OR      ^code(P) and (< > CR) (< > DR)
           POS     Closed (<F>) )
; Consome mensagem nao esperada da fila

(TR T14
   PRE     WFUR (<F>)
   SINC    UCEP ? ConResp
   SINC    MCEP ! UnitReq(Pdu ^code CC)
   POS     Open (<F>) )

(TR T15
   PRE     WFUR (<F>)
   SINC    UCEP ? DisReq
   SINC    MCEP ! UnitReq(Pdu ^code DR)
   POS     Closing (<F>) )

; FASE DE TRANSFERENCIA DE DADOS

; Apenas algunas transicoes foram modeladas visando
obter um modelo simulavel. As transicoes referentes a
parte de troca de dados nao foram representadas.

(TR T17
   PRE     Open (<F>)
   SINC    MCEP ? UnitInd(P)
   POS     PlaceOpen (<P>) )

(TR T25
   PRE     PlaceOpen (<P>)
   COND    ^code(P) = DR
           SINC UCEP ! DisInd
           POS  SemiClosed (<F>)
   OR      ^code(P) = CR
           SINC    Mcep !
                   UnitReq(Pdu ^code CC)
           POS Open(<F>)
   OR      ^code(P) and (< > DR) (< > CR)
           POS Error(<F>) )

; FASE DE LIBERACAO DA CONEXAO

(TR T28
   PRE     Open (<F>)
   SINC    UCEP ? DisReq
   SINC    MCEP ! UnitReq(Pdu ^code DR)
   POS     Closing (<F>)
           @N-attempts = 0 )

(TR T29
   PRE     Closing (<F>)
   SINC    MCEP ? UnitInd(P)
   POS     PlaceClosing (<P>) )

(TR T30
   PRE     PlaceClosing (<P>)
   COND    ^code(P) or (= DR) (= DC)
           POS  Closed (<F>)
   OR      ^code(P) and (< > DR) (< > DC)
           POS  Closing (<F>) )

(TR T32
   PRE     Closing (<F>)
   COND    @N-attempts < @N
           DELAY (@P, @P)
           POS     DelayClosing (<F>)
           @N-attempts = (+ 1 @N-attempts)
   OR      @N-attempts >= @N
           POS     Closed (<F>) )

(TR T34
   PRE     DelayClosing (<F>)
   SINC    MCEP ! UnitReq (Pdu ^code DR)
   POS     Closing (<F>) )

(TR T35
   PRE     SemiClosed (<F>)
   SINC    MCEP ! UnitReq(Pdu ^code DC)
   POS     Closed (<F>) )

; FASE DE ERRO

(TR T36
   PRE     Error (<F>)
   SINC    UCEP ! DisInd
   POS     SavingError (<F>) )

(TR T37
   PRE     SavingError (<F>)
   SINC    MCEP ! UnitReq(Pdu ^code DR)
   POS     Closed (<F>)
           @N-attempts = 0 )

; MARCACAO INICIAL DA REDE

(INIT  Closed ( (Ficha) ) )

) ;EndAbraEntity
```

```
(MODULE Medium
(CLASS (Pdu ^posicao ^code ^seq ^data) )
(VAR P  (Pdu) )
(PORTS
   Mcep_0    IN    (UnitReq(P))
             OUT (UnitInd(P))
   Mcep_1    IN    (UnitReq(P))
             OUT (UnitInd(P)) )
(MODULE FilaFifo
(CLASS
   (Fila ^fifo)
   (Pdu ^posicao ^code ^seq ^data) )
(VAR
   Q (Fila)
   P  (Pdu) )
(PORTS
   IpIn          IN       (UnitReq(P))
   IpOut         OUT      (UnitInd(P)) )
(PLACES  L1 (<Q>)
         L2 (<P>) )
(TR TIns
   PRE    L1 (<Q>)
   SINC   IpIn ? UnitReq(P)
   POS    L1 (<Q>)
          L2 (<P>)
          ^posicao(P) =
           (+ 1 (ultimo ^fifo(Q)))
          ^fifo(Q) =
           (insere ^posicao(P) ^fifo(Q) )
)
(TR TRem
   PRE    L1 (<Q>)
          L2 (<P>)
          ^posicao(P) =
           (primeiro ^fifo(Q))
   SINC   IpOut ! UnitInd(P)
   POS    L1 (<Q>)
          ^fifo(Q) = (resto ^fifo(Q) )
)
(INIT L1 ( (Fila ^fifo ()) ) )
)
(INSTANCE FilaA WITH FilaFifo)
(INSTANCE FilaB WITH FilaFifo)
(ATTACH   Mcep_0 TO
          FilaA.IpIn FilaB.IpOut)
(ATTACH   Mcep_1 TO
          FilaB.IpIn FilaA.IpOut)
) ;EndMedium
```

```
(MODULE User
(CLASS
      (Sdu ^data) )
(VAR
      S       (Sdu) )
(PORTS
      UCEP  IN (    ConInd
                    ConConf
                    DisInd
                    DatInd (S) )
            OUT (ConRec
                    ConResp
                    DisReq
                    DatReq (S) ) )
```

; Como somente nos interessa estudar o comportamento das Entidades de Protocolos (AbraEntity) em resposta a solicitações do usuário (User), as transições do módulo User são consideradas sempre sensibilizadas e são cada uma delas responsáveis pelo envio/recepção das unidades de serviço especificadas na declaração de porto acima.

```
)
(INSTANCE EntityA WITH AbraEntity)
(INSTANCE EntityB WITH AbraEntity)
(INSTANCE UserA WITH User)
(INSTANCE UserB WITH User)
(INSTANCE Meio WITH Medium)
(CONNECT  UserA.UCEP TO EntityA.UCEP)
(CONNECT  UserB.UCEP TO EntityB.UCEP)
(CONNECT  EntityA.MCEP TO Meio.MCEP_0)
(CONNECT  EntityB.MCEP TO Meio.MCEP_1)
)

(de ultimo (l)
     (cond   ((null l) 0)
             (t (car (last l)))))

(de insere (s l)
     (cond   ((null l) (cons s ()))
             (t (cons (car l) (insere s (cdr l))))))

(de primeiro (l)
     (car l))

(de resto (l)
     (cdr l))
```

APÊNDICE 3

FORMA GRÁFICA DA RPO MODELANDO CADA FASE DOPROTOCOLO ABRACADABRA

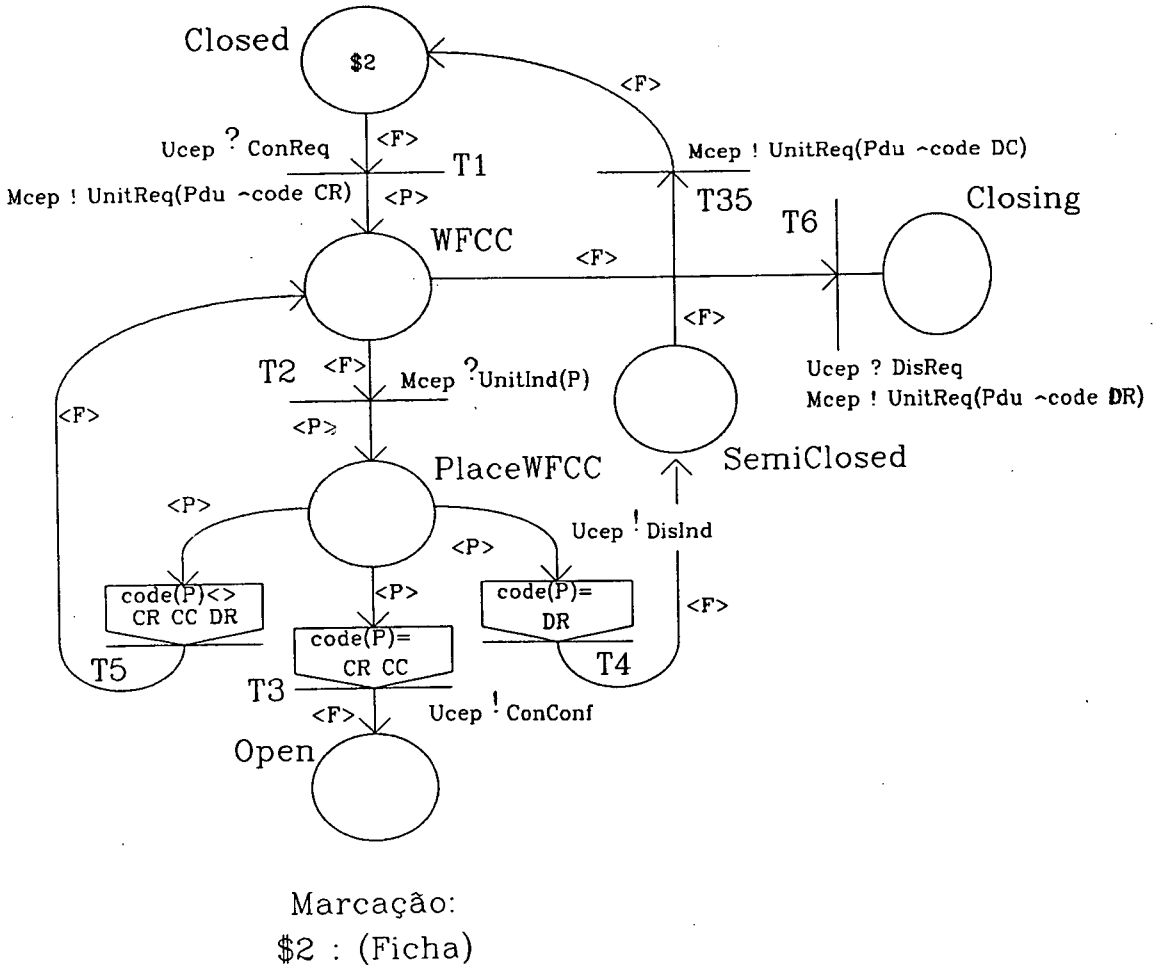


Fig. Ap. 1. Representação em RPO do iniciador da conexão.

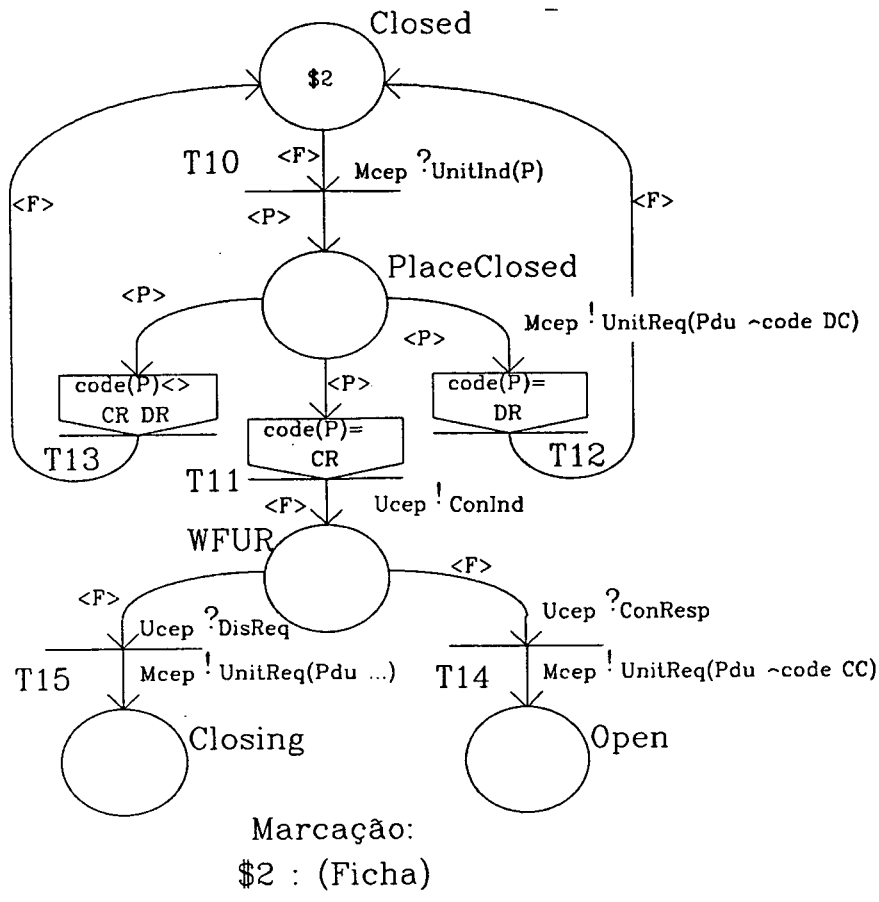


Fig. Ap. 2. Representação em RPO da resposta a um pedido de conexão.

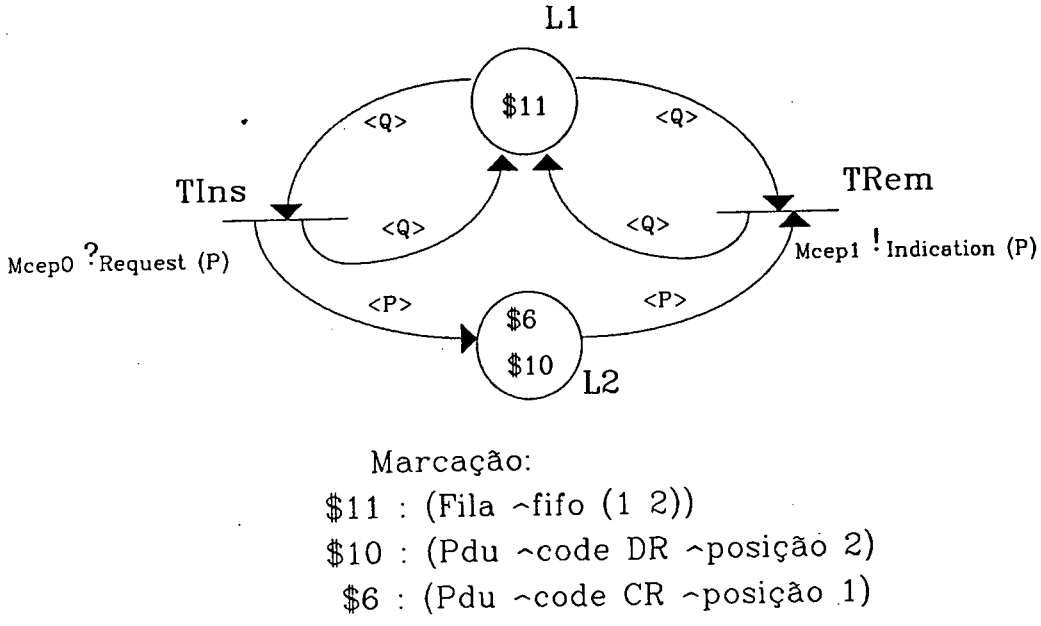


Fig. Ap. 3. Representação em RPO da fila FIFO.

APÊNDICE 4

RESULTADOS SIMULAÇÃO

AP.4.1. Célula Flexível de Usinagem

```
rpo> (load cfu)
* Carregada rede: cfu

rpo> (prior)
* Prioridade: t

rpo> (m)
* Marcacao Corrente (profundidade = 0):
 p1
   $8 : (Peca ^ident p2 ^job (m2 m1))
   $7 : (Peca ^ident p1 ^job (m1 m2 m1))
 p3
   $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
   $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
 p2
   $4 : (Pallet ^posicao 4 ^estado nil)
   $3 : (Pallet ^posicao 3 ^estado nil)
   $2 : (Pallet ^posicao 2 ^estado nil)
   $1 : (Pallet ^posicao 1 ^estado nil)

rpo> (tr)
* Transicoes do modelo
 entrada
 saida
 rotacao
 inicio
 fim

rpo> (d)
* Transicoes Sensibilizadas
1 : entrada : ($8 $1)  {0}
2 : entrada : ($7 $1)  {0}
3 : rotacao : ($1 $2 $3 $4)  {1}
Entre sua escolha ou 0 se nao deseja disparos: 1
  Disparada: entrada : ($8 $1)

rpo> (d)
* Transicoes Sensibilizadas
1 : rotacao : ($1 $2 $3 $4)  {1}
Entre sua escolha ou 0 se nao deseja disparos: 1
  Disparada: rotacao : ($1 $2 $3 $4)

rpo> (run 9)
11

rpo> (path)
* Caminho desde profundidade 0
  0 : ((entrada ($8 $1)))  : 1
  1 : ((rotacao ($1 $2 $3 $4)))  : 2
  2 : ((entrada ($7 $4)))  : 3
  3 : ((rotacao ($4 $1 $2 $3)))  : 4
  4 : ((inicio ($4 $7 $5)))  : 5
  5 : ((fim ($4 $7 $5)))  : 6
  6 : ((inicio ($1 $8 $6)))  : 7
  7 : ((fim ($1 $8 $6)))  : 8
  8 : ((rotacao ($3 $4 $1 $2)))  : 9
  9 : ((inicio ($4 $7 $6)))  : 10
 10 : ((fim ($4 $7 $6)))  : 11

rpo> (m)
* Marcacao Corrente (profundidade = 11):
 p2
   $8 : (Peca ^ident p2 ^job (m1))
   $7 : (Peca ^ident p1 ^job (m1))
   $4 : (Pallet ^posicao 3 ^estado p1)
   $3 : (Pallet ^posicao 2 ^estado nil)
   $2 : (Pallet ^posicao 1 ^estado nil)
   $1 : (Pallet ^posicao 4 ^estado p2)
 p3
   $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
   $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)

rpo> (memory)
* Memoriza estado de interesse
   1 - Visualizar
   2 - Memorizar
   3 - Voltar
 Entre com sua escolha: 2
   Defina um nome para o estado: est1

rpo> (bt 4)
4

rpo> (prior)
* Prioridade: ()

rpo> (run 7)
11
```

```
rpo> (path)
* Caminho desde profundidade 0
  0 : ((entrada ($8 $1)))  : 1
  1 : ((rotacao ($1 $2 $3 $4)))  : 2
  2 : ((entrada ($7 $4)))  : 3
  3 : ((rotacao ($4 $1 $2 $3)))  : 4
  4 : ((inicio ($4 $7 $5)))  : 5
  5 : ((fim ($4 $7 $5)))  : 6
  6 : ((inicio ($1 $8 $6)))  : 7
  7 : ((fim ($1 $8 $6)))  : 8
  8 : ((rotacao ($3 $4 $1 $2)))  : 9
  9 : ((rotacao ($4 $1 $2 $3)))  : 10
 10 : ((rotacao ($1 $2 $3 $4)))  : 11

rpo> (m)
* Marcacao Corrente (profundidade = 11):
 p2
    $8 : (Peca ^ident p2 ^job (m1))
    $7 : (Peca ^ident p1 ^job (m2 m1))
    $4 : (Pallet ^posicao 3 ^estado p1)
    $3 : (Pallet ^posicao 2 ^estado nil)
    $2 : (Pallet ^posicao 1 ^estado nil)
    $1 : (Pallet ^posicao 4 ^estado p2)
 p3
    $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
    $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)

rpo> (memory)
* Memoriza estado de interesse
   1 - Visualizar
   2 - Memorizar
   3 - Voltar
  Entre com sua escolha: 3
    Voltar para qual estado: est1

rpo> (prior)
* Prioridade: t

rpo> (m)
* Marcacao Corrente (profundidade = 11):
 p2
    $8 : (Peca ^ident p2 ^job (m1))
    $7 : (Peca ^ident p1 ^job (m1))
    $4 : (Pallet ^posicao 3 ^estado p1)
    $3 : (Pallet ^posicao 2 ^estado nil)
    $2 : (Pallet ^posicao 1 ^estado nil)
    $1 : (Pallet ^posicao 4 ^estado p2)
 p3
    $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
    $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)

rpo> (run 11)
22

rpo> (m)
* Marcacao Corrente (profundidade = 22):
 p1
    $8 : (Peca ^ident p2 ^job ())
    $7 : (Peca ^ident p1 ^job ())
 p3
    $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
    $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
 p2
    $4 : (Pallet ^posicao 4 ^estado nil)
    $3 : (Pallet ^posicao 3 ^estado nil)
    $2 : (Pallet ^posicao 2 ^estado nil)
    $1 : (Pallet ^posicao 1 ^estado nil)

rpo> (run 5)
27

rpo> (path)
* Caminho desde profundidade 0
  0 : ((entrada ($8 $1)))  : 1
  1 : ((rotacao ($1 $2 $3 $4)))  : 2
  2 : ((entrada ($7 $4)))  : 3
  3 : ((rotacao ($4 $1 $2 $3)))  : 4
  4 : ((inicio ($4 $7 $5)))  : 5
  5 : ((fim ($4 $7 $5)))  : 6
  6 : ((inicio ($1 $8 $6)))  : 7
  7 : ((fim ($1 $8 $6)))  : 8
  8 : ((rotacao ($3 $4 $1 $2)))  : 9
  9 : ((inicio ($4 $7 $6)))  : 10
 10 : ((fim ($4 $7 $6)))  : 11
 11 : ((rotacao ($2 $3 $4 $1)))  : 12
 12 : ((rotacao ($1 $2 $3 $4)))  : 13
 13 : ((inicio ($1 $8 $5)))  : 14
 14 : ((fim ($1 $8 $5)))  : 15
 15 : ((rotacao ($4 $1 $2 $3)))  : 16
 16 : ((inicio ($4 $7 $5)))  : 17
 17 : ((fim ($4 $7 $5)))  : 18
 18 : ((rotacao ($3 $4 $1 $2)))  : 19
 19 : ((saida ($1 $8)))  : 20
 20 : ((rotacao ($2 $3 $4 $1)))  : 21
 21 : ((saida ($4 $7)))  : 22
 22 : ((rotacao ($1 $2 $3 $4)))  : 23
 23 : ((rotacao ($4 $1 $2 $3)))  : 24
 24 : ((rotacao ($3 $4 $1 $2)))  : 25
 25 : ((rotacao ($2 $3 $4 $1)))  : 26
 26 : ((rotacao ($1 $2 $3 $4)))  : 27
```

```
rpo> (init)
* Marcacao Corrente (profundidade = 0):
 p1
   $8 : (Peca ^ident p2 ^job (m2 m1))
   $7 : (Peca ^ident p1 ^job (m1 m2 m1))
 p3
   $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
   $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
 p2
   $4 : (Pallet ^posicao 4 ^estado nil)
   $3 : (Pallet ^posicao 3 ^estado nil)
   $2 : (Pallet ^posicao 2 ^estado nil)
   $1 : (Pallet ^posicao 1 ^estado nil)

rpo> (del $7)

rpo> (make p1 (Peca ^ident p3 ^job (m1 m1)))

rpo> (make p1 (Peca ^ident p4 ^job (m2 m2)))

rpo> (m)
* Marcacao Corrente (profundidade = 0):
 p1
  $10 : (Peca ^ident p4 ^job (m2 m2))
   $9 : (Peca ^ident p3 ^job (m1 m1))
   $8 : (Peca ^ident p2 ^job (m2 m1))
 p3
   $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
   $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
 p2
   $4 : (Pallet ^posicao 4 ^estado nil)
   $3 : (Pallet ^posicao 3 ^estado nil)
   $2 : (Pallet ^posicao 2 ^estado nil)
   $1 : (Pallet ^posicao 1 ^estado nil)

rpo> (s)
* Transicoes Sensibilizadas
1 : entrada : ($10 $1)  {0}
2 : entrada : ($9 $1)  {0}
3 : entrada : ($8 $1)  {0}
4 : rotacao : ($1 $2 $3 $4)  {1}

rpo> (bt -1)
0

rpo> (m)
* Marcacao Corrente (profundidade = 0):
 p1
   $9 : (Peca ^ident p3 ^job (m1 m1))
   $8 : (Peca ^ident p2 ^job (m2 m1))
 p3
   $6 : (Maquina ^nome m2 ^posicao 3 ^estado nil)
   $5 : (Maquina ^nome m1 ^posicao 2 ^estado nil)
 p2
   $4 : (Pallet ^posicao 4 ^estado nil)
   $3 : (Pallet ^posicao 3 ^estado nil)
   $2 : (Pallet ^posicao 2 ^estado nil)
   $1 : (Pallet ^posicao 1 ^estado nil)

rpo> (obj $7)
$7 : (Peca ^ident p1 ^job (m1 m2 m1))
```

AP. 4.2. Protocolo Abracadabra

```
; Exemplo detalhado da Fase de Abertura de
; Conexao, iniciada pela EntityA e respondida
; afirmativamente pela EntityB

rpo> (load abracadabra)
* Carregada rede abracadabra

rpo> (m)
* Marcacao Corrente (profundidade = 0):
Modulo: EA
 Closed
   $1 : (Ficha)
Modulo: EB
 Closed
   $2 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UA.ConReq  <->  EA.T1' : ($1)
2 : UB.ConReq  <->  EB.T1' : ($2)
Entre sua escolha ou 0 se nao deseja disparos: 1
  Disparada: UA.ConReq  <->  EA.T1' : ($1)

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UB.ConReq  <->  EB.T1' : ($2)
2 : EA.T1'' ($1) <-> Meio.FilaA.TIns ($3)
Entre sua escolha ou 0 se nao deseja disparos: 2
  Disparada: EA.T1'' ($1) <-> Meio.FilaA.TIns ($3)

rpo> (m)
* Marcacao Corrente (profundidade = 2):
Modulo: EA
 WFCC
   $1 : (Ficha)
Modulo: EB
 Closed
   $2 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo (1))
 L2
   $5 : (Pdu ^code CR ^posicao 1)
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())
```

```
rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UB.ConReq  <->  EB.T1' : ($2)
2 : Meio.FilaA.TRem ($3 $5) <-> EB.T10 ($2)
3 : UA.DisRec  <-> EA.T6' ($1)
Entre sua escolha ou 0 se nao deseja disparos: 2
 Disparada: Meio.FilaA.TRem ($3 $5) <-> EB.T10
($2)

rpo> (m)
* Marcacao Corrente (profundidade = 3):
Modulo: EA
 WFCC
   $1 : (Ficha)
Modulo: EB
 PlaceClosed
   $5 : (Pdu ^code CR)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : EB.T11 ($5)  <->  UB.ConInd
2 : UA.DisRec <-> EA.T6' ($1)
Entre sua escolha ou 0 se nao deseja disparos: 1
 Disparada: EB.T11 ($5)  <->  UB.ConInd

rpo> (m)
* Marcacao Corrente (profundidade = 4):
Modulo: EA
 WFCC
   $1 : (Ficha)
Modulo: EB
 WFUR
   $6 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UB.ConResp  <->  EB.T14' ($6)
2 : UB.DisReq  <->  EB.T15' ($6)
3 : UA.DisRec  <-> EA.T6' ($1)
Entre sua escolha ou 0 se nao deseja disparos: 1
 Disparada: UB.ConResp  <->  EB.T14' ($6)

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UA.DisRec <-> EA.T6' ($1)
2 : EB.T14" ($6) <-> Meio.FilaB.TIns ($4)
Entre sua escolha ou 0 se nao deseja disparos: 2
 Disparada: EB.T14" ($6) <-> Meio.FilaB.TIns ($4)

rpo> (m)
* Marcacao Corrente (profundidade = 6):
Modulo: EA
 WFCC
   $1 : (Ficha)
Modulo: EB
 Open
   $6 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo (1))
 L2
   $7 : (Pdu ^code CC ^posicao 1)

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UA.DisRec <-> EA.T6' ($1)
2 : Meio.FilaB.TRem ($4 $7) <-> EA.T2 ($1)
3 : UB.DisReq <-> EB.T28 ($6)
Entre sua escolha ou 0 se nao deseja disparos: 2
 Disparada: Meio.FilaB.TRem ($4 $7) <-> EA.T2
($1)

rpo> (m)
* Marcacao Corrente (profundidade = 7):
Modulo: EA
 PlaceWFCC
   $7 : (Pdu ^code CC)
Modulo: EB
 Open
   $6 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())

rpo> (d)
* Transicoes Sensibilizadas
* Pares Sincronizados c/ "rendez-vous"
1 : UB.DisReq <-> EB.T28 ($6)
2 : EA.T3 ($7) <-> UA.ConConf
Entre sua escolha ou 0 se nao deseja disparos: 2
 Disparada: EA.T3 ($7) <-> UA.ConConf
```

```
rpo> (m)
* Marcacao Corrente (profundidade = 8):
Modulo: EA
 Open
   $8 : (Ficha)
Modulo: EB
 Open
   $6 : (Ficha)
Modulo: Meio.FilaA
 L1
   $3 : (Fila ^fifo ())
Modulo: Meio.FilaB
 L1
   $4 : (Fila ^fifo ())

rpo> (path)
* Caminho desde profundidade 0
 0 : ((UA.ConReq)  (EA.T1' : ($1))) : 1
 1 : ((EA.T1" ($1))  (Meio.FilaA.TIns ($3))) : 2
 2 : ((Meio.FilaA.TRem ($3 $5))  (EB.T10 ($2))) : 3
 3 : ((EB.T11 ($5))  (UB.ConInd)) : 4
 4 : ((UB.ConResp))  (EB.T14' ($6))) : 5
 5 : ((EB.T14" ($6))  (Meio.FilaB.TIns ($4))) : 6
 6 : ((Meio.FilaB.TRem ($4 $7)))  (EA.T2 ($1))) : 7
 7 : ((EA.T3 ($7))  (UA.ConConf)) : 8
```

```
; Exemplo de outros tracos verificados na
simulacao,
; relativos a Fase de Abertura de Conexao

rpo> (path)
* Caminho desde profundidade 0
 0 : ((UA.ConReq)  (EA.T1' : ($1))) : 1
 1 : ((EA.T1" ($1))  (Meio.FilaA.TIns ($3))) : 2
 2 : ((Meio.FilaA.TRem ($3 $5))  (EB.T10 ($2))) : 3
 3 : ((EB.T11 ($5))  (UB.ConInd)) : 4
 4 : ((UB.DisReq))  (EB.T15' ($6))) : 5
 5 : ((EB.T15" ($6))  (Meio.FilaB.TIns ($4))) : 6
 6 : ((Meio.FilaB.TRem ($4 $8)))  (EA.T2 ($1))) : 7
 7 : ((EA.T3 ($8))  (UA.DisInd)) : 8
 8 : ((EA.T35 ($9)) (Meio.FilaA.TIns ($3))) : 8
 9 : ((Meio.FilaA.TRem ($3 $10)) (EB.T30 ($10))) :

rpo> (Obj $8 $9 $10)
$8  : (Pdu ^code DR)
$9  : (Ficha)
$10 : (Pdu ^code DC)
```

```
rpo> (path)
* Caminho desde profundidade 0
 0 : ((UA.ConReq)  (EA.T1' : ($1))) : 1
 1 : ((UB.ConReq)  (EB.T1' : ($2))) : 2
 2 : ((EB.T1" ($2))  (Meio.FilaB.TIns ($4))) : 3
 3 : ((EA.T1" ($1))  (Meio.FilaA.TIns ($3))) : 4
 5 : ((Meio.FilaA.TRem ($3 $11))  (EB.T2 ($2))) : 6
 6 : ((Meio.FilaB.TRem ($4 $12))  (EA.T2 ($1))) : 7
 7 : ((EB.T3 ($12))  (UB.ConConf)) : 8
 8 : ((EA.T3 ($11))  (UA.ConConf)) : 9

rpo> (Obj $11 $12)
$11 : (Pdu ^code CR)
$12 : (Pdu ^code CR)
```